Supporting Adaptive Flexibility with Communications Middleware

Dirk van der Linden, Georg Neugschwandtner,
Maarten Reekmans, Herbert Peremans
University of Antwerp
Belgium
{*dirk.vanderlinden, georg.neugschwandtner,*
maarten.reekmans, herbert.peremans}*@uantwerpen.be*

Wolfgang Kastner
Automation Systems Group
Vienna University of Technology
Austria
k@auto.tuwien.ac.at

***Abstract*—Automation systems are continuously growing in scope and size. To keep them maintainable (despite their ever-increasing complexity), structures, methodologies and technologies with the capability of responding to diverse and changing requirements are required – a quality we call adaptive flexibility. After presenting highlights from a survey illustrating the variety of requirements, this paper discusses two approaches to supporting adaptive flexibility as well as the relationship between them. The first is OPC Unified Architecture (UA), a communications middleware standard. The second is the Normalized Systems Theory, a formal approach to ensuring systems evolvability. The paper intends to bridge the gap between theory and practice by highlighting several aspects of OPC UA that support adaptive flexibility, with this analysis being based on the concepts of Normalized Systems as far as possible.**

***Keywords*-Automation; OPC UA; Profiles; Scalability; Diversity; Modularity; Reusability; Evolvability; Normalized Systems.**

I. Introduction

Industrial communication has become a key technology in modern industry. A continually growing number of manufacturing companies desire, even require, totally integrated systems. This integration should cover electronic automation devices such as Programmable Logic Controllers (PLCs) and microcontrollers as well as Human Machine Interfaces (HMI) and supervision, trending, and alarm software applications, e.g., Supervisory Control and Data Acquisition (SCADA) and Manufacturing Execution Systems (MES). Industrial communication encompasses the entire range from field device and controller to manufacturing operations management and Enterprise Resource Planning (ERP) applications.

Likewise, the past decade has seen a push towards the integration of building services and building management. Total integration in this field should not only cover Direct Digital Control (DDC) and SCADA/Building Management Systems (BMS), but also Computer Aided Facility Management (CAFM) applications and HMI ranging from dedicated panels and visitor guidance systems to webbased solutions on tablets and smartphones.

Such "totally integrated systems" are not monolithic or developed from scratch, but consist of multiple (sub)systems – such as those just mentioned – connected to form a (more or less) coherent whole. Connecting independent subsystems, which were developed independently, can be a veritable challenge. On the other hand, exactly this separation into independent subsystems is one of the best ways to deal with the high overall complexity of an integrated system. Thus, how a large system can be split into subsystems or modules on the one hand and how these can be connected on the other hand are topics worth exploring.

Modularity is the foundation for several desirable properties, including reusability as well as:

Scalability – the possibility to adapt the configuration of a system in order to fulfil demanding requirements but not be oversized for less demanding requirements,

Diversity – the freedom to choose between (and/or accommodate) different implementations of a particular function, and

Evolvability – the ability of a system to follow as requirements change with time, and stay maintainable.

These aspects are interrelated. For example, a system which supports diversity with regard to a particular subfunction will evolve more gracefully when another, independently implemented, instance of this subfunction needs to be merged into the system (consider, for example, adding a second printer to a PC). In the following, we will refer to all these aspects using the umbrella term *adaptive flexibility* – the ability of a system to adapt to (changing or diverse) requirements.

Adaptive flexibility is an essential quality since, generally speaking, "one size fits all" solutions do not exist – or do not really fit. There is a reason why so many specialized kinds of systems have developed: in the world of industry (and beyond), companies specialize in different tasks. These different tasks come with specific technical requirements, and companies approach them with different solutions. This variety of requirements and approaches is illustrated by the results of a survey we performed [1]. Requirements and preferred approaches are also changing over time. Thus, there is clearly a need for structures, methodologies and

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

technologies that are capable of supporting customization and change. Modularization is a key concept in this regard. For maximum benefit in terms of adaptive flexibility, modules (or subsystems) must be decoupled as thoroughly as possible. On the other hand, they must interoperate properly. Both are considerable and well known challenges.

Adaptive flexibility and interoperability are commonly considered valuable goals in software engineering practice. Designers of methodologies and standards typically use an intuitive approach to support these goals. OPC UA is a recent communications middleware standard, which – as will be shown below – incorporates several related design choices. OPC initially stood for "OLE for Process Control", but the current OPC UA (Unified Architecture) specifications are no longer based on OLE. OPC UA is also popular in practice: two European developer and user conferences in 2012 and 2013 gathered around 150 attendees each. Taking another angle, the Normalized Systems Theory (NST) [2] is an example for a formal approach to support adaptive flexibility, more precisely, evolvability. Its goal is to provide formal rules on how to construct evolvable software programs, instead of relying on heuristic knowledge for this purpose. This theory was developed with software architectures for business applications in mind, but has been successfully applied to other domains such as industrial control and business processes [3], [4].

This paper intends to bridge the gap between theory and practice. It examines several aspects of OPC UA which support adaptive flexibility. As far as possible, this analysis is based on the concepts laid forth by NST. This makes our evaluation of OPC UA more stringent by providing a theoretical foundation. In addition, it illustrates NST concepts by putting them in the context of a concrete implementation. Given the substantial size of the OPC UA specifications (over a thousand pages in total), this endeavour has to be explorative in nature and limited to highlighting selected aspects. The paper extends previous work which focused on the above mentioned survey and on a recommender tool relating OPC UA specification feature sets to requirements [1]. Other previous work considered the application of NST to automation systems from various angles: with respect to couplings and dependencies between subsystems [5], the design of evolvable, modular PLC programs [6], and the separation of input/output and control functions in such programs [7]. Together with insights on how some design qualities heralded by NST were intuitively built into web technology [8], this inspired the expanded discussion of the relationship between OPC UA and NST concepts which can be found in this paper.

First, OPC UA is introduced, including the profile mechanism to support scalability. Then, NST and its goals are summarized. Section IV discusses the results of our worldwide survey, showing the wide variety in application requirements and technologies. Section V considers how connecting applications via OPC UA middleware can increase adaptive flexibility. NST makes certain stipulations regarding the interface between modules. This section presents examples of why some of these stipulations are satisfied, and how others can be satisfied, when this interface is based on OPC UA. Section VI examines which internal mechanisms of OPC UA support adaptive flexibility, looking at these mechanisms from a black box perspective. Finally, Section VII uses the OPC UA stack and services as a backdrop and concrete example to illustrate how finely system implementations must be divided into modules according to NST. Section VIII concludes the paper.

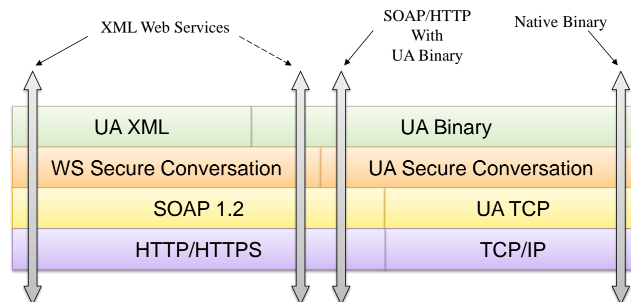

Figure 1. OPC UA transport [11]

II. OPC UNIFIED ARCHITECTURE

The OPC Foundation started in the mid-1990s to promote cross-vendor interoperability for automation projects. Initially, the OPC specifications were based on Distributed Component Object Model (DCOM) as a communication technology. DCOM is Microsoft's proprietary technology used for communication between software modules distributed across networked computers. The more recent standard family, OPC Unified Architecture (UA), is designed to be more generic, abstract, technology independent and platform agnostic [9], [10]. OPC UA is based on a cross-platform Service Oriented Architecture (SOA) and includes security mechanisms. Its two fundamental components are mechanisms for data transport on one hand and data modelling on the other hand.

The OPC UA specification contains abstract definitions of OPC UA services for data communication on the application level. Mapping these services to a concrete technology, the transport mechanisms tackle platform independent communication while still allowing optimisation with regard to the involved systems. Currently, OPC UA defines two transport mappings that are used for establishing a connection between an OPC UA client and server on the network level. UA/TCP is fast and simple and SOAP/HTTP is firewall-friendly and uses Web Services (WS). While communication between industrial controllers or embedded systems may require high performance and low overhead, business management applications may need an easily parsed data format. As a

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

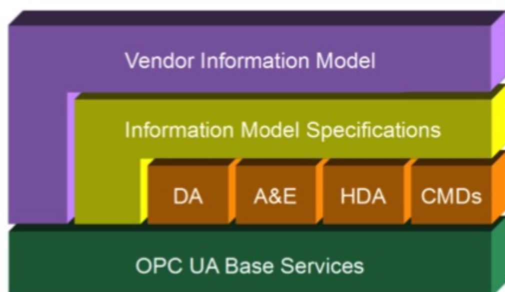

Figure 2. OPC UA information model domains [12]

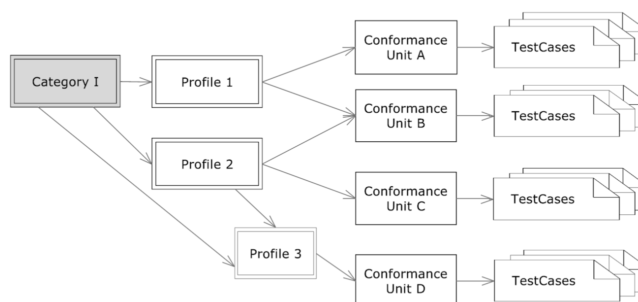

Figure 3. OPC UA Profiles and ConformanceUnits [1]

consequence, two data encoding schemes are defined, called OPC UA Binary and OPC UA XML. Different compromises are possible to find a good balance between security and performance, depending on the application (Figure 1).

The objective of an OPC UA server is to present information of an underlying (automation) process so that it can be used to seamlessly integrate with other processes and management systems. The exposed information represents the current, and possibly the historic state and behaviour of the underlying process. OPC UA defines rules and basic building blocks to expose such an information model. Basically, an OPC UA information model is made up of nodes and references that represent the relationship between nodes. Nodes can contain both online data (instances) and meta data (classes). OPC UA clients can browse through the nodes of an OPC UA server via the references, and gather data from and information about the underlying system.

The OPC Foundation provides dedicated OPC UA information models to structure the legacy OPC specifications (Figure 2). These information models support common tasks of legacy OPC interfaces [13]. These legacy interfaces are data access (DA), alarms and events (A&E), historical data access (HDA) and commands (CMDs). By modelling them with OPC UA, the transition from legacy systems to the new OPC UA communication standard is made easier. In addition, the OPC Foundation encourages definitions of complex data based on related industrial standards. Examples are IEC 61131-3 (PLC programming languages [14]), FDI (Field Device Integration) with EDDL (Electronic Device Description Language) [15] and ISA 95 (integration of enterprise and factory automation and control systems) [16]. Client software can be conveniently written against these complex data types. They also increase the potential of code re-use.

OPC UA is designed in a way that individual implementations do not need to support all features, but can be downscaled to a limited scope if desired. At the same time, advanced products which allow a high degree of freedom will require the support of more sophisticated features. A service based OPC UA implementation can be tailored to be just as complex as needed for the underlying application.

Hence, what is needed is a way to describe (and test) which features are supported by an OPC UA compliant product. A specific set of features (e.g., a set of services or a part of an information model) that can be tested as a single entity is referred to as a *ConformanceUnit*. An example of a ConformanceUnit is “Method Call”, containing the call service that is used to call a method on an OPC UA server. ConformanceUnits are further combined into *Profiles*. An application (client or server) shall implement all of the ConformanceUnits in a profile to be compliant with it. Some profiles may contain optional ConformanceUnits, which in turn may exist in more than one profile (Figure 3). Software certificates contain information about the supported profiles. OPC UA Clients and Servers can exchange these certificates via services.

Up to now, more than 60 OPC UA Profiles have been released [17]. The number of released profiles is continuously being extended by OPC Foundation working groups. It is expected that, over time, also other organisations will take part in this activity.

III. Normalized Systems

In general, software gradually becomes unmaintainable as features are added over time. The theory of Normalized Systems (NST) [18] proposes an approach to counter this effect. According to [2],

> *...Normalized Systems are (information) 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.*

A. Software is aging

Let us consider the effort necessary to modify system S according to a change requirement (e.g., to add a feature). This requires an effort that depends on the change required (Figure 4).

Following Parnas, software is aging [19]. There are two, quite distinct, types of software aging. The first is caused by the failure of the product’s owners to modify it to meet

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

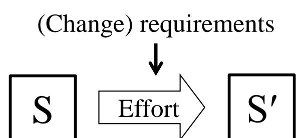

Figure 4. System evolution

changing needs; the second is the result of the changes that are made. Lehman stated that this second cause of aging is a decay of structure and formulated the law of increasing complexity [20]:

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

Thus, the effort that must be spent on a change does not only depend on the change required; in addition, it increases with time. The authors of the Normalized Systems Theory (NST) combine Lehman's law of increasing complexity with the assumption of unlimited systems evolution [3]:

> *The system evolves for an infinite amount of time, and consequently the total number of requirements and their dependencies will become unbounded.*

They admit that, in practice, this assumption is an overstatement for most commercial applications. However, it provides a theoretic view on the evolvability issue, which is independent of time. If we combine this assumption with the law of Lehman, we see that, over time, the impact of required changes will become unbounded in terms of the effort to implement them.

It is challenging to determine the detailed cause of this deterioration. Which new parts of the system contribute to the effect described by this law? In other words, why does adding a feature to the code cause more costs in the *mature* stage of the lifecycle of a system than adding exactly the same feature in the *beginning* stage of the project?

The challenge the authors of NST want to take is to keep the impact of a change dependent on the nature of the change itself, not on the size (or amount of changed or added requirements) of the system. In other words, they want to keep this impact *bounded.* The rather vague questions like "Is this change causing more troubles than another?" should be replaced by the fundamental question: "Is this change causing an unbounded effect?". The authors of NST want to provide a deterministic and unambiguous yes/no answer to this question, by evaluation whether one of the NST theorems is violated or not.

Conversely to a change with bounded impact, changes causing impacts that are dependent on the size of the system are called *combinatorial* effects in NST. Systems where changes do not cause combinatorial effects are called 'stable.' *Normalized Systems* are stable in this sense. 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 5). By eliminating combinatorial effects, this relation can be kept linear for an unlimited period of time, and an unlimited amount of changes to the system. In other words, to achieve stability, combinatorial effects must be removed from the system.

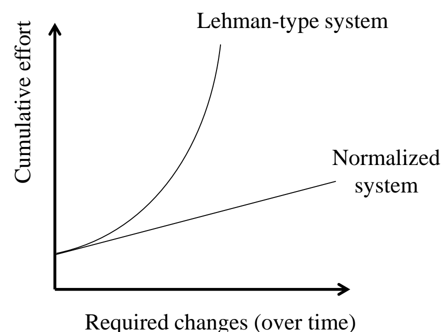

Figure 5. Cumulative change effort over time

The shape of the Lehman curve in Figure 5 is a function of the amount of combinatorial effects in the system, which again depends on the tacit knowledge of the developers and/or software architects. Since tacit knowledge cannot be measured in exact numbers by definition, it is not possible to give a mathematical definition of the shape of the Lehman curve. However, the curve surely becomes flatter when the experience or tacit knowledge of the developer rises. Indeed, a well-performed maintenance activity or 're-write' will reduce the combinatorial effects within the system or subsystem (visible in Figure 6 as discontinuities along the y-axis). In such a 're-write' activity, no extra functional requirements is added. Rather, the structure is improved in a heuristic way, involving tacit knowledge.

There is a limit to improving the shape of the Lehman curve by applying tacit knowledge. While the concepts of NST are not completely new, they make existing heuristic, "tacit" knowledge explicit. This way, it becomes possible to group and apply the (formerly) tacit knowledge of several experts, with the eventual result of reaching the goal of bounded impact.

B. Anticipated changes

Listing up all functional requirements, including those already present but not yet uncovered and those which may come up in the future, is an overly ambitious endeavour. Indeed, numerous system analysts have found that this task is impossible in an ever changing technical and economical environment. The authors of NST do not state they can do better. Instead, they propose to make use of *anticipated*

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

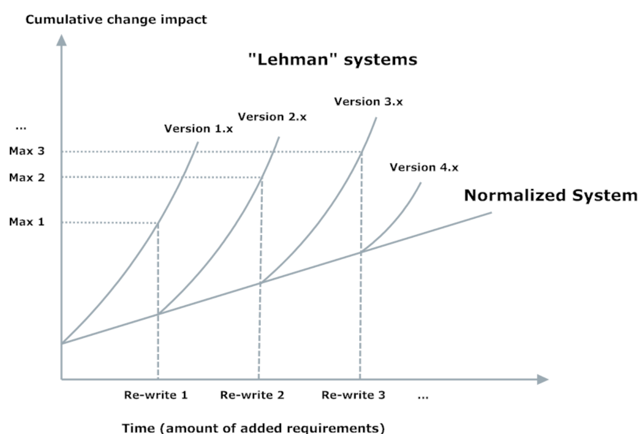

Figure 6. Reduction of cumulative effort by way of a rewrite [8]

changes. These changes are not directly associated to change or feature requests expressed by customers or managers. Instead, anticipated changes focus on *elementary* changes to software primitives: action entities, which are modules containing functionality, and data entities, which are sets of tags or fields. In this way, NST does not focus on complicated high-level changes as a whole, but on elementary changes performed on data and action entities. Typically, one real-life change corresponds to a number of elementary changes.

The set of anticipated (elementary) changes is as follows:

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

System changes to meet "high-level requirements" that are obtained by system analysts by traditional gathering techniques (including interviews and use cases) [18] should be converted to these abstract, elementary anticipated changes. We were able to convert all high level changes to one or more of these abstract anticipated changes in several case studies [21], [6], [7].

As an example of breaking down a high-level change into anticipated changes, consider a compressed air installation whose capacity must be increased. The installation initially consists of a single compressor. The control code for this compressor contains data fields like start and stop commands, run or failure states, or manual/automatic states and commands – a data entity – and the necessary logic to implement the appropriate behaviour associated with these data fields – an action entity.

- The change requires the addition of a second compressor. This second compressor also requires control code like the first one: this implies an instance of the anticipated change "an additional action entity" and an instance of the anticipated change "an additional data entity."
- When the new compressed air installation does not reach the desired pressure after a configurable amount of time with the original compressor only, the second shall be started in cascade. Adding this cascade logic implies an instance of the anticipated change "an additional action entity." To make sure the wear and tear is equally divided between both compressors, a permutation algorithm is implemented: at the first run, the first compressor starts, and after the next downtime the second compressor starts first – and vice versa. This permutation logic again implies an instance of the anticipated change "an additional action entity," but it also implies "an additional data entity," because the compressor which was started last has to be remembered.
- A new version of the code module that calls the original control module of the original compressor must be created to ensure that both compressors, cascade and permutation logic are encapsulated. This implies an instance of the anticipated change "a new version of an action entity" as well as an instance of the anticipated change "a new version of a data entity", because the underlying new data has to be encapsulated as well.
- Finally, we need to ensure version transparency, which means that both the old and the new version have to be supported. Indeed, this logic might be used for several compressed air installations, from which several coexisting versions need to be maintained. This implies an instance of the anticipated change "a new version of an action entity".

C. Design theorems for Normalized Systems

The design theorems or principles of Normalized Systems, i.e., systems that are stable with respect to the above set of elementary changes, are:

1) Separation of concerns: An action entity can only contain a single task in Normalized Systems.

This principle is focusing on how tasks are implemented within processing functions. Every concern or task has to be separated from other concerns – a concept with a long tradition, including Dijkstra using the term in an essay he wrote in 1974 [22]. In this way, one can focus on one aspect at a time. Tasks are identified based on change drivers: a task is something which is subject to an independent change. Whenever two or more pieces of functionality in a module can be anticipated to change independently of each other, they must be reassigned to separate modules. Section VII contains an example for a change driver to illustrate the concept.

2) Data version transparency: Data entities that are received as input or produced as output by action entities need to exhibit version transparency in Normalized Systems.

Reducing component incompatibilities between current and previous versions is a relevant research topic in software engineering (see, e.g., [23]). The version transparency theorems contribute towards addressing this challenge. Data version transparency is related to how data structures are passed to processing functions. The requirement of data version transparency is fulfilled when data entities can exist in multiple versions, without affecting the processing functions that consume or produce them. In other words, an old data entity should contain a version number, so that any functionality in a module can recognize its 'age' and tolerate that newer data fields are missing. Conversely, a new data entity should keep the fields from older versions, so that older action entities do not need to be aware of the newer fields.

3) Action version transparency: Action entities that are called by other action entities need to exhibit version transparency in Normalized Systems.

This principle is focusing on how processing functions are called by other processing functions. Action version transparency is the property that action entities can have multiple versions without affecting any of the other processing functions which call this processing function. In other words, when an older action entity calls a younger one, the younger action entity should process the call as if it would be as old as the calling action entity. Conversely, when a younger action entity calls a older module, the younger entity should expect a response corresponding to the older version.

4) Separation of states: The calling of an action entity by another action entity needs to exhibit state keeping in Normalized Systems.

This principle is focusing on how calls between processing functions are handled. The contribution of state keeping to stability is based on the removal of coupling between modules that is due to errors or exceptions. Per call, the caller should maintain a separate data entity to track the state of this call. When an action entity calls another action entity, it should not block to wait for the response of the called module, or even worse, block forever if the response is not like expected. For example, when the response message is of a newer version, which is unknown to an older calling module, the calling module should treat the response as 'unknown', rather than being blocked until the expected response arrives.

D. Versions vs. variants

The above discussion considers versions in the context of "older" and "younger" data and action entities. However, this principle can also be used to achieve a related property of evolvability: support for diversity. Here again, modules are related from a functional perspective; again, they are different versions of the same core task. However, in the case of diversity, these versions do not have a consecutive character. Instead, they are more like alternative implementations of the same task. As an example, consider different brands of variable frequency drives having different tag names, tag data types or slightly different functionality. In this context of diversity, we suggest to consider "versions" as "variants" instead.

E. Encapsulation of software entities

In Normalized Systems, the elementary action and data entities are very small. On the level of applications, there is a need for larger elements with more comprehensive functionality. To achive this, the elementary entities can be encapsulated. Different types of encapsulation are suggested [21], including:

- Action Encapsulation: It is important to be aware of the core functionality of a module. Following the separation of concerns principle, this core functionality should be separated from supporting functionality, because the supporting functionality can evolve independently from the core functionality and vice versa. Action encapsulation ensures that the core functionality and the supporting functionality are kept together, without hampering the independent evolution of the composed entities. One entity for the core task (core action entity) is surrounded by entities for supporting tasks (supporting action entities). Together, they can form an action element.
- Data Encapsulation: The arguments of the individual action entities within an action element, i.e., a number of data entities, can be encapsulated as a data element for use by the action element. The data corresponding to the functionality of the action element must be structured with regard to the action entities which the action element contains. Any action entity can read all data of all the other action entities, but, for each specific set of data, there is only one action entity which is allowed to modify it. In other words, an individual data entity corresponds to an individual action entity with regard to the permission to modify or manipulate the data. All other data entities are, for this particular action entity, read-only. In other words, the data containing a data element is composed of data entities, which are separated with regard to the permission to modify or manipulate; still, all information relevant for the entire action element is kept together in one place for reading.

Encapsulation can also be recursive: elements may in turn contain elements. In addition to action entities and data entities, NST defines flow entities, trigger entities and connection entities. Encapsulation also applies here: based on these entities, flow elements, trigger elements and connection elements can be created. Flow entities combine action entities into a sequence and act as a container for the execution states of these action entities. Depending on these states, trigger entities decide whether an action element

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

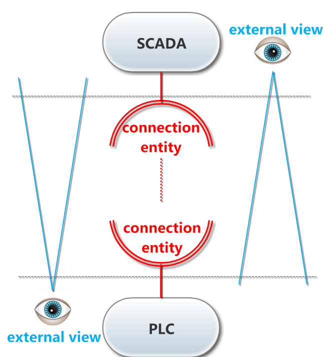

Figure 7. PLC and SCADA de-coupled by a connection entity

has to be triggered. Connection entities and elements are especially relevant in this paper and will be discussed below.

F. Connection encapsulation and migration

The separation of concerns principle imposes that the use of an external technology in an action entity implies an extra, separated task or construct, as it is possible that the external technology will evolve differently from the background technology environment of the action entity.

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. As soon as there is no control about the evolution of an external system from the view of a Normalized System, such an external system has to be treated as a separate concern. The connection between the external system and the Normalized System is made by way of a *connection entity*. In case of an update of the external system, a new eonnection entity corresponding to the new version is added. A connection element encapsulates these connection entities and selects the one which represents the appropriate version.

As an example, Figure 7 shows a connection entity placed between a PLC and a SCADA system. De-coupling subsystems by using connection entities is an essential step to applying NST in practice: The Normalized Systems Theory promotes a high granularity of (sub)systems. It is certainly a challenging task to make existing systems, which do not have such a high granularity, comply with NST. For these systems, there is a need for a migration path towards Normalized Systems like we recognized in earlier work [8].

Modularity is in general accepted as a good engineering practice [24]. Consequently, Lehman systems typically are implemented respecting a form of modularity, albeit not granular enough to comply with the separation of concerns principle. Nevertheless, we emphasize that having a modular (Lehman) system, constructed based on rather large modules or subsystems, might be a valuable start of a migration path towards the achievement of a NST. In such a scenario,

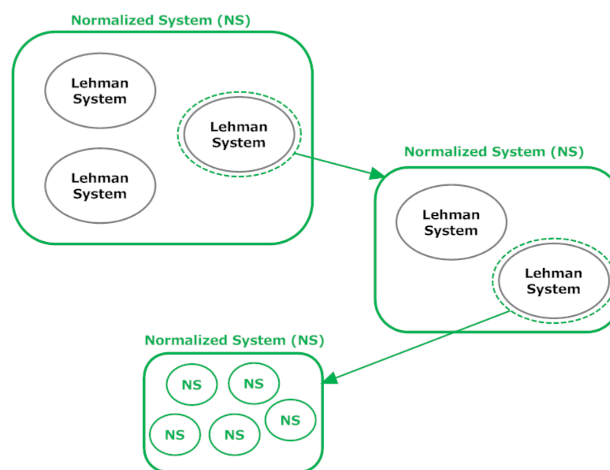

Figure 8. Migration from Lehman to Normalized subsystems [8]

one can concentrate on one single subsystem, which has an amount of couplings with the other subsystems. The first step in the migration scenario is to isolate this subsystem by inserting connection entities into each individual coupling (as illustrated in Figure 7). After isolating this subsystem, one can independently further rewrite the subsystem step-by-step towards removing all internal combinatorial effects, while the connection entities prevent these internal combinatorial effects to have an impact on the other subsystems 'outside'. Figure 8 visualizes this stepwise "fencing off" of Lehman systems.

IV. Requirements Translation

In order to understand where and how to apply adaptive flexibility, it is important to understand the type of industrial applications and how they evolve. We also need an indication of the scalability requirements of those applications and the diversity of the components used in modern day industrial applications.

OPC UA profiles can be considered modular building blocks from which OPC UA servers and clients can be constructed. Typically certain OPC UA profiles, the ones that provide the core functionality, will be used in almost all servers and clients. Other profiles will only be used for applications that require specific functionality provided by these particular profiles. Knowing which profiles are used for a certain type of application provides valuable information regarding the situations in which adaptive flexibility can be applied successfully. Currently, the choice of OPC UA profiles is pragmatically made according to the implementor's knowledge of the OPC UA specifications and the perceived requirements of the application. New concepts of OPC UA, for example information modelling, redundancy or events tend to be skipped by implementors due to a lack of awareness of these profiles.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Knowing the diversity, scalability and evolvability of industrial applications is relevant to our research, but on top of that it would be valuable to determine which OPC UA profiles can be recommended for certain branches of industry. Such a recommendation would give the application architect and project manager a guideline to deciding which profiles are most relevant to implement. This approach follows the assumption that each industry sector requires specific automation applications, resulting in a typical set of automation technologies being used and, likewise, having typical requirements on data communication within and between these technologies. Knowing which OPC UA profile (or combination thereof) is designed to fulfil given communication requirements, it should be possible to recommend a set of profiles based on the industry sector. For example, the redundancy profiles can be recommended for sectors like chemical industry, where high availability is important. Traceability is important for the pharmaceutical sector, so the Auditing profiles would be included in the recommended profiles list.

A. Worldwide survey

We designed a survey [1] to validate this assumption. The survey did not assume any detailed knowledge of OPC UA profiles on the part of the respondents, but focused on generalised questions regarding communication requirements that would allow drawing conclusions about required profiles. To make sure that these questions reflect the capabilities of the available OPC UA profiles well, we consulted one of the lead authors of the Profiles part of the OPC UA specifications for expert advice.

By analysing the results of this survey we can distill valuable information about the diversity of the applications used in modern day industrial systems. The survey should also give an indication of the requirements of scalability in these types of applications. How the requirements evolve over time cannot be seen from the survey results.

To address a representative number and kind of stakeholders, the survey was distributed to OPC Foundation members as well as companies that figure on the Foundation's regular mailing list and several other industry specific mailing lists containing a wide variety of respondents in addition.

About 25,000 questionnaires were sent out, and a total of 719 responses were collected. The geographical distribution of all respondents is shown in Figure 9. It largely matches the geographical distribution of the OPC Foundation members.

Respondents were asked to specify the industrial sectors they are active in. Multiple answers were allowed. The sectors which yielded 15% or more of responses are listed in Table I. A significant number (10–15%) of respondents reported activity in up to 8 different industrial sectors, and still 5–10% are represented in up to 5 areas.

Table I
INDUSTRY SECTORS MOST RELEVANT TO RESPONDENTS [1]

Sector	%
Oil & Gas Production	18%
Oil & Gas Distribution	15%
Chemical	18%
Food & Beverage	16%
Power Distribution	16%
Power Generation	20%
Building Automation	19%
Automotive Industry	16%
Industrial Automation	38%
Process Automation	30%
IT	19%

Table II
TECHNOLOGIES IN USE BY RESPONDENTS [1]

ERP	MES	SCADA	PLC	PAC	DCS	TFM	BMS	DDC
30%	24%	66%	72%	29%	43%	9%	18%	13%

When looking at the technologies reported to be used by respondents (Table II; DCS = Distributed Control System, PAC = Programmable Automation Controller, TFM = Technical Facility Management), we see that respondents clearly indicate PLC and SCADA as dominant automation technologies. Other technologies are also reported to be used extensively in combination with the dominant technologies. This is an indication that a wide variety of systems and subsystems are present in the companies questioned in this survey. Also, quite general management tasks such as alarm, event and user logging received high importance rankings among respondents.

Thus, while there is apparently the need for support of a diverse range of different systems, initial implementation effort can be significantly reduced by focusing on these technologies. Considering that many applications follow a very basic pattern, many implementers will only need to provide the so called *Core Server* profile, in combination with one *Transport* profile.

We found some key trends and assumptions behind the OPC UA technology that can be confirmed by the survey results. For example, 432 interviewees stated to be manu-

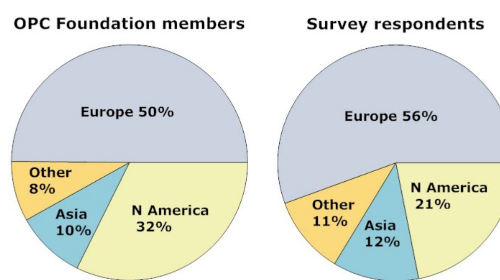

Figure 9. OPC Foundation members and respondents by region [1]

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

facturer of systems or products that use a communication network. 59% use a field device network, and 37% use the control network in a shared network set-up with the standard computer network. This illustrates the high importance of industrial data communications in general as well as the drive towards combined communication networks and totally integrated systems.

As far as the speed of communication is concerned, communication within less than one second is required by the majority (165/355) of PLC/PAC/DCS users. Also, the time frame for delivering data in the control network is typically short (15% say less than 1 ms; 55% say less than one second). However, a substantial percentage of PLC/PAC/DCS users (81/355) are satisfied with a delivery of data/messages within less than one minute.

This shows that on one hand, demand for fast and efficient transport as provided by *UA-TCP UA-SC UA Binary* transport profile is significant. On the other hand, a market segment exists where lower speed may well be acceptable if compensated by other desirable properties such as firewall friendly communication and an easily parsed data format, which would for example be a key property of the *SOAP-HTTP WS-SC XML* transport profile.

There is also a strong demand for security and robustness. The top three security related issues among respondents are authentication, restricted access and confidentiality of transferred data; for availability, utilizing redundant servers is seen as more relevant than deploying redundant clients.

Regarding operating systems and programming languages in use by the respondents, a technology shift begins to show. Though Windows is still the leading operating system being deployed, a trend towards Linux can be observed. Relevant programming languages are, in decreasing order of importance, C/C++, C#.NET, VB.NET, and Java. The rise of .NET indicates that DCOM is becoming a legacy technology. The use of C#.NET and C/C++ is significantly higher than the other languages ($p < 0.001$). Differences concerning the use of the programming languages in different regions are not significant (at a p-value of 0.05), which leads us to conclude that the technology shift is happening worldwide.

To confirm the suspected dependencies between industry sectors, automation technologies and communication requirements, we applied logistic regression analysis [25] to the survey results. In such an analysis, the estimates of the weight of variables with regards to a specific use provides an idea of the relevance of these variables. The results of our analysis are described in the following.

We found the use of MES to be very high in the food and beverage industry. PLC systems are being used nearly everywhere except in power distribution and IT (with negative estimates of -0.54 and -0.89, respectively). The use of DCS systems is also very diverse, except in the automotive industry, which instead shows a significant use of PAC (at an estimate of 0.70). SCADA is present in power generation, industrial automation, food/beverage and oil production, with a negative estimate for the IT sector (-0.64). Overall, PLC and SCADA are quite correlated (0.55).

Again, using logistic regression analysis, we found differences of preferences of programming languages with regard to the type of automation technology in use. The majority of Java users can be found among ERP, MES, SCADA and TFM users (as confirmed by the Hosmer and Lemeshow Goodness-of-Fit test). The majority of C#.NET users work, in decreasing order, with MES, SCADA and ERP systems. The majority of C users focus on SCADA, DDC and BMS systems. The diversity of VB users is the biggest, they work with PLC, SCADA, MES, DCS and ERP systems. The selection of these technologies is based on the analysis of maximum likelihood estimates of a simplified model with an entry cutoff value of 0.15 and a stay cutoff value of 0.15.

Concerning the most common security issues, we found a good fitting logistic regression model showing that ERP users value rogue system detection, auditability of actions, confidentiality of proprietary data and network intrusion avoidance. PLC users have different priorities, with a focus on auditability of actions, availability of systems, restricted external access to proprietary data and network intrusion avoidance. PAC users place a similar (but lower) priority on network intrusion avoidance, availability of systems and restricted external access. MES users assign high importance to preventing the alteration of proprietary data, auditability of actions, network intrusion detection and authentication of users.

The users who need a very short time frame (less than 1 ms) for delivering data/messages via the control network are mostly MES and PLC users. Those who need the fastest message exchange via the computer network (less than one second) are mostly PLC, MES and SCADA users.

We see that many companies use a lot of different target technologies and have very diverse communication requirements. There is no straightforward correlation between the industry sectors when looking at these communication requirements and technologies. A simple static set of recommended profiles according to industry branches is therefore not apparent. So, we abandoned the intention of making an industry specific, static set of OPC UA profiles.

We can see that the activities of each company are very diverse, almost independent of the industry sector. Each company uses its own approach and mix of technologies, tailored to its very diverse requirements. We focused on finding new ways to provide a clear recommendation of OPC UA profiles, taking into account the individual diverse requirements of each company.

B. Role-based optimization strategy

After processing the survey and analysing the results, we opted to approach the problem of translating requirements into profiles in another way: an online tool to dynamically

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

produce recommended sets of profiles on an individual, user-by-user basis. The tool is based on the generalized questions regarding communication requirements that reflect the capabilities of the various OPC UA profiles that were created for the survey. It is designed to easily accept new or updated questions to reflect newly released profiles. This *agility* is an additional advantage, as the definition of OPC UA profiles by the OPC Foundation working groups is an ongoing process.

Considering our initial goal of identifying which specific sets of profiles would add the most value for a specific company, we wanted to have the tool take into consideration the economic dimension in addition to the technical one. Each vendor has its own target market, with an assorted set of customers and specific fields of application. While many profiles might make sense from a technical point of view (and thus may well all be requested by customers), implementing some profiles will provide more commercial benefits than implementing others. Vendors must meet the challenge to find the balance between satisfying customer requirements and return on investment for implementing these profiles. To best support this decision, our tool should therefore assign a priority to each recommended profile. Also, it should be capable of linking an estimate of commercial benefits based on development time and budget to this prioritized list of recommended profiles. With this information, end-users of the tool can more accurately envision the development planning of a product even without detailed knowledge of OPC UA technologies, as this knowledge is embedded in the tool.

Thus, for getting the most relevant results, the implementation of the decision support tool takes into account normative constraints (i.e., it shall produce output that is consistent with the OPC UA specifications), budget constraints and maximum commercial benefit.

The decision support tool takes its input from three sources, each representing a particular competence or role. These inputs provide the functional parameters for the decision support tool. When the experts have entered these parameters, the end-user, who typically has little knowledge of OPC UA profiles, can use the tool to help determine the list of recommended profiles for their company and application.

The first role is that of an OPC UA expert who is determining the normative constraints. The main task of the UA expert is to input a set of survey questions and possible answers. Each answer is then linked to one or more profiles. Using these relations a profile is produced according to the answer given by a respondent to the respective question. Besides, what we call *static* normative constraints have been hardcoded into the software. Some examples of these static normative constraints are that no product can be built with only one profile and that an application must at least support one of the core profiles, one security profile and one transport profile. Another example of a static normative constraint relates to nested profiles: the basic profile must be implemented before an enhanced profile can be implemented (e.g., *Core Server* can only be implemented when *SecurityPolicy - None* has already been implemented).

The second role is that of a software architect who provides input regarding the development time required for implementing a specific profile. The software architect must have detailed knowledge of OPC UA profiles to do this. The development time is put into the tool once per profile. The end-user has to provide some additional parameters like the cost of programming labour in their company, the preferred programming language and an indication of the complexity of the application behind the OPC UA interface to get the total cost of implementation of a specific profile.

Third, the role of technical-commercial manager (sales / business) is to estimate the commercial benefit of implementing a specific profile. This commercial benefit can be estimated and used as a parameter to manage the development priority. Some of the commercial benefits can be estimated by the results of our technology survey. As mentioned, it should be noted that typically the technical-commercial role does not have enough OPC UA knowledge to estimate the benefit of a profile directly, which means that they especially profit from decision support as described in this section.

The tool was implemented and put online on a private page. We contacted one of the main contributors of the OPC UA specification part that defines the different profiles. He would take on the role of OPC UA expert. We ourselves also took on this role to come up with questions to which the answer reflects what profiles are relevant. For some basic profiles, the input of the tool is straightforward. For example, a question about redundancy needs in a company:

Does your company use any of the following? (select the ones that apply)

- Redundant servers
- Redundant clients
- Redundant communication devices
- Fault tolerant systems
- I don't know

This question polls how much need there is for the following profiles:

- Redundancy Switch Client Facet
- Redundant Client Facet
- Redundancy Transparent Server Facet
- Redundancy Visible Server Facet

With straightforward profiles questions, the answers given by the respondent can be clearly linked to certain profiles. The translation of basic requirements for communication into a list of profiles that are recommended to be implemented for the respondent was achieved by inputting these

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

questions and answers into the tool. There is also a mechanism implemented to assign priorities to the recommended profiles. The profiles with a higher priority are considered to be more important and more relevant to implement. So, the first important goal that we set for the tool, delivering a list of recommended profiles, is met.

Obtaining the cost of implementing a profile is one of the aspects of the tool that proved very difficult. Implementation is something specific to each company and a diverse set of parameters determines that cost. For example, in a Lehman system, adding a profile to a very simple application will require less effort than adding a profile to a complex application. Also, the programming language and possible libraries used and the experience of the developers are parameters that determine the economic impact of adding a certain profile. The cost estimation feature was not added in this version of the tool. For Lehman systems, it is by definition not possible to predict the actual costs of adding a component. So, the cost of adding a profile to an existing application, not written according to NST, can only be an estimation. We decided that the input provided by a software architect cannot result in a cost estimation that is accurate enough to be valuable information for the users of the tool.

While evaluating the usefulness of the tool, we approached several key people from the OPC Foundation and several developers, integrators and managers. The first testers of our tool reported a positive experience and saw potential in its outcome. However, the fact that not all profiles were included was reported as a problem. When validating the tool further, it did not seem to have the anticipated resonance and adoption in the OPC community. The tool can provide valuable information about basic functionality, but because of the diversity of the applications this version of the tool can not provide conclusive results.

As an outcome of the work done analysing the results of the survey and creating the tool, we can say that a large diversity of components and sub-components can be seen in industrial applications. We can also conclude that the technologies used in the industry typically have to scale well. To approach the problem of keeping these diverse applications maintainable and scalable means focusing on fundamental concepts and methodologies like adaptive flexibility. Adaptive flexibility can be achieved by applying the concepts known from NST, which have already been successfully used in business software. For creating industrial applications according to the principles of NST, OPC UA technology can provide significant support.

V. OPC UA FOR CONNECTION ENCAPSULATION

OPC UA is dedicated to providing interoperable – industrial – communication. The real production or business application functionality remains a subject of customized implementation. Implementers of this functionality can use OPC UA as an enabler to interoperate with other applications. From the perspective of these applications, OPC UA can be seen as an external technology itself; also, it provides access to other external vendor-specific or platform-specific technologies. In other words, OPC UA is an excellent match for the idea of separating technologies by way of separated tasks, as indicated in the separation of concerns principle. Again, consider Figure 7: OPC UA is excellently suited to acting as the connection entity. This subsection focuses less on the internal structure of OPC UA, but rather on the way how an OPC UA interface can block combinatorial effects and represent continuously evolving applications and/or data sources.

Following Lehman's law of increasing complexity, complexity increases and maintainability decreases when the size of a system grows. This is visualized in Figure 5. System integrators typically do not build all components of applications from scratch, but they use available commercial products as a part of their solutions. They rarely have access to the source code of these components, neither do they have the resources available to rewrite them. Therefore, it is not possible for them to make the entire solution comply with the NST theorems. However, if they manage to isolate the components from each other – or block combinatorial effects from propagating between them –, they have a better chance of staying close to the origin on the Lehman curve of subsystems (Figure 5). In other words, they can reduce the impact of combinatorial effects by keeping the size of the subsystems small. This situation is similar to our proposed migration scenario in Figure 8.

While OPC UA middleware could be implemented as a Normalized System, this is not likely to occur soon. Despite the fact that NST is derived from heuristic knowledge, it is very unlikely that developers who are not aware and familiar with it can implement a fully evolvable system. However, OPC UA does separate applications, and block combinatorial effects. Note that a distributed system, composed of a diversity of applications, interconnected with OPC UA, is actually not in conformance with the NST theorems, at least not if we regard this distributed system as a whole – it is not "fully normalized". Still, OPC UA can be a valuable tool to separate the subsystems. In other words, an OPC UA layer can separate Lehman systems, while being a Lehman system itself (and, thus, not complying with the NST theorems).

Consider, for example, an evolution scenario where two smaller systems should be interconnected to form a larger system. The larger system should support the diversity introduced by integrating these two systems, while allowing each of the original two systems to continue evolving independently. Expanding on this example, consider that a module of the first system should be reused in the second system – it will be challenging to keep this module compatible with both systems while they continue to evolve independently. We think it is possible to meet this challenge when (sub)systems

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

are based on OPC UA.

A. OPC UA as an integration bus

In the early nineties, the development efforts to access automation data in devices increased while the market was becoming more and more global. The number of different bus systems, protocols and interfaces used to access automation data was called 'uncountable' [26]. In that period, this access was typically based on so-called product-specific drivers. In [27], a product-specific driver is defined as follows:

> *Product-specific drivers describe software components that have been developed for a specific product. They are linked to this product so that they cannot be used with products of other manufacturers. These drivers make available data in a manufacturing-specific form.*

When building solutions in a cross-vendor environment, the 'uncountable' number of product-specific drivers becomes problematic. Addressing this problem was the main motivation of the OPC Foundation task force (1994) to develop a connectivity standard, which became the (classic) OPC standard.

In [2], the use of a messaging or integration bus is stated to be a manifestation of the separation of concerns principle. Replacing product-specific drivers with an OPC interface fulfils the purpose of an integration bus. In a cross-vendor environment, accessing a vendor's product through a coupling using a product-specific driver can be seen as a violation of the separation of concerns principle. Indeed, when an amount of SCADA products are available on the world market, the introduction of any new type of PLC (or PLC access protocol) requires all these SCADA products to be updated in order to be able to support this new type. As the amount of available products increases, the impact of this combinatorial effect becomes worse.

B. Version transparency in the address space

When two (sub)systems are connected via OPC UA, the server arranges data related to its application functionality in the OPC UA address space, where the client can access it. Generally speaking, the OPC UA address space is a collection of nodes and references between those nodes; in simple cases, it takes the form of variable values arranged in a tree structure. However, OPC UA supports advanced data modelling in the address space, including complex objects, method invocation, and type hierarchies.

A change in the server system does not affect how the OPC UA interface works. Neither does it necessarily affect where the client can find data from the server system on this interface: the server need not reflect every change in its address space. As long as the change is not relevant to the part of the system exposed by the server, it will be completely hidden from the client. This does not only apply to changes due to system evolution over time, but also to differences due to diversity. The standard information models (for example, [14], [16]) further strengthen this concept, making the representation of certain data vendor independent.

The structure of an OPC UA address space is very flexible, which allows various ways of supporting evolvability. We can obtain version transparency by simply limiting the changes we allow a server to make to its address space over time: no nodes should be modified or deleted. Rather, the evolution of an address space should be based on additions only. This way, a client which is not aware of the change can continue accessing older existing properties, attributes and references and ignore the new information.

If necessary, the server could also place a new version of its data in a new branch of its address space, while continuing to maintain the information in the original branch for non-agile clients, which will in turn continue to access the data in the way they were developed. If the client is more recent than the server in such a scenario, the "older" server will inform the client that the branch holding the new version does not exist in its address space by answering the client request with the StatusCode BadInvalidArgument. In addition, the server could also place information about the versions it supports at a well-known place in its address space, taking advantage of the flexible structure of the OPC UA address space. Either way, both versions can coexist, whether in-place or in separate parts of the address space.

The previous discussion assumes that the client disposes of pre-configured information about which data to expect where in the server address space. In use cases where this does not apply, the standardized meta information in the OPC UA information model can provide considerable benefit. Object types allow a client to recognize instances of entities in previously unknown places. In such a scenario, type hierarchies enable diversity and version transparency. For example, a client can interact with an unknown motor controller as long as it conforms to a known supertype (representing a basic motor controller interface). The client can identify the instance in the address space by following the HasTypeDefinition and HasSubtype references.

OPC UA also standardizes metadata related to address space versions which a server has the option to expose. The NodeVersion attribute of a Node changes when a node or reference is added or deleted (for example, when a variable node is added below an object node) or when the DataType attribute of a variable or VariableType changes. Clients should be aware of this change: in case of a deletion or data type modification, they may otherwise obtain wrong data. This can be prevented by continually checking the NodeVersion for changes; however, such an approach is not efficient in terms of communication. Therefore, OPC UA specifies the ModelChangeEvent to inform clients that they should expire their node cache and rebrowse the relevant

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

part of the address space. Related to this mechanism are the DataTypeVersion attribute, which gives information about changes to the definition of data types with custom encoding, and the SemanticChangeEvent for changes to Value Attributes of Properties. These changes also need to be monitored by the client, but are not covered by NodeVersion and ModelChangeEvent, respectively.

Note that if the address space evolves in a version transparent way, following the theorems of Normalized Systems, the mechanisms outlined in the previous paragraph are not required: since previous versions remain accessible to the client, there is no need to be aware of model changes. As a practical consideration, however, a process similar to garbage collection will be needed in the long run in case one wants to delete obsolete nodes, properties or attributes. This process would need to be based on warranties that these entities will no longer be used by any client.

Finally, OPC UA also allows accessing historical address space. This is implemented on top of the Views concept, which allows to present clients with subsets of the address space tailored to a specific purpose (e.g., access to maintenance relevant data only). The ViewVersion parameter in the Query services enables a client to browse previous versions of the address space, including nodes which have in the meantime been deleted. This allows accessing historical variable data attached to such nodes, which would otherwise no longer be reachable. While this mechanism does not appear to be intended to enable access to current data via an older interface (i.e., View) version, nothing would prevent a server to update Variable Nodes in the historical address space with current values and thus use this mechanism for the purpose of version transparency.

C. Profiles and dependencies

The concept of modularity is most commonly associated with the process of subdividing a system into several subsystems [28]. This decomposition allows modifications at the level of a single subsystem instead of having to adapt the whole system at once [24]. To achieve the ideal form of modularity of a system, the underlying subsystems should be loosely coupled and independent [29]. If some dependencies are inevitable, they should be described and made explicit for the user of the subsystem. This enables the user to decide whether to satisfy the dependency or to not use the subsystem.

As an example, consider a program which requires a particular runtime library: you may decide to add the library to your system, or you may rather use another program if the required library would be in conflict with a library version you have in use (and the library management in your system cannot handle this situation). Another example, but from a different application domain, would be a multiplexer component to be placed on a new integrated circuit (IC): for providing its logic function, it depends on the particular IC manufacturing process it was designed for.

A subsystem behind an OPC UA interface is accessed in the form of services provided over a network. The dependencies, therefore, are:

1) A network stack (in the current specifications, TCP/IP).
2) A library implementing the OPC UA protocol. With OPC UA, this is usually comprised of a “stack” for lower level functionality and an “SDK” (Software Development Kit) between this stack and the application.
3) Application logic to interact with the address space (via the SDK: reading/writing values, update notifications, ...).
4) Application logic to map application data structures into the address space.

The network connection between the OPC UA server and client may be a machine-local loopback interface; a physical network interface is required when server and client run on different computers.

As already discussed in previous sections, the OPC UA protocol is comprehensive and complex. The Stack, SDK and application are free to only use a subset of it. Dependencies 2 and 3 in the list above therefore scale to the requirements of the application. The OPC UA specification groups related functionality into profiles. Profile implementations can be modules – units of functionality. Since it is possible to select and implement only a subset of them, this is a manifestation of the separation of concerns principle: if they would not address separate concerns, one would be forced to implement all of them.

It is possible for an OPC UA client and server to have different sets of profiles implemented, but still be able to communicate. This also means that a client or server can have various communication partners that each support a different profile subset. OPC UA contains mechanisms to enable meaningful communication by selecting matching profile subsets for both communication partners. The mechanisms allow one subsystem (the client) to select the best way to interact with another subsystem (the server), based on which profiles each counterpart supports. This is a manifestation of version transparency at the level of communicating applications. It is also a way of exhibiting diversity – supporting different versions at the same time.

One group of profiles concerns transport functionality, with the ability to choose between UA/TCP or SOAP/HTTP, OPC UA Binary or OPC UA XML, and various levels of communication security (security policies). The client can query the server for available endpoints before attempting to establish a communication channel. Each of these endpoints corresponds to a transport configuration; they will depend on the capabilities of the server stack and SDK as well as the server configuration (it may choose to not offer, for example, insecure communication for policy reasons

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

although stack and SDK would have the ability). The client then checks which of the endpoints matches its capabilities and preferences best and uses this endpoint to establish a secure channel and session. This mechanism enables the server to clearly document its requirements – for example, in terms of acceptable connection security; the client can decide to satisfy the resulting dependency by making the necessary functionality available on its side (for example, via an appropriate library) or decide to not use the server.

Once the session is established, the client has various options how to interact with the server address space. The OPC UA specifications require that a server always supports the most basic set of operations, which are sufficient to retrieve a list of related profiles implemented by the server from a well-known place in its address space. Each of these profiles corresponds to a set of services supported by the server, for example, whether it supports alarms and conditions or if it allows a client to modify the address space. The profile documents which services can be invoked on the server, and in which ways they can be invoked.

The client is not forced to use any of this “special” functionality unless it chooses to. In case it does, matching client profiles are described in part 7 of the OPC UA specifications, describing the dependencies in terms of protocol functionality which the client must provide on its side to take advantage of these server features. In addition, the client can determine from the list of profiles supported by the server which services it may invoke without receiving an error message. To ensure that client and server stacks and SDKs from different developers will communicate flawlessly as long as they implement a compatible set of client and server profiles, the OPC Foundation has established a compliance and certification program for interoperability testing.

Profiles are identified by unique Uniform Resource Identifiers (URI). If the functionality associated with a profile is modified in the future, the updated version will again receive a unique URI, reflecting the new version. This means that profile versions can coexist.

Thanks to the mechanisms and documentation described, an OPC UA client can gather comprehensive information about the dependencies which are caused by its wish to communicate (or communicate efficiently) with a server. No dependencies are hidden; the OPC UA server is a true “black box” in the sense of [29]. It is fully specified in terms of an outside view on its functionality; a client should never have to examine how the server is implemented to be able to use its services. The OPC Foundation also maintains a “profile reporting” website [17] which is documenting the functionality associated with profiles (again, in terms of OPC UA services, not their implementation).

However, these mechanisms and documentation only cover functionality related to the OPC UA interface. By design, OPC UA profiles do not describe anything related to the actual application functionality “behind” the OPC UA interface (although standardized information models such as [14] or [16] blur this distinction somewhat). An example for such a dependency may be that the subsystem requires periodic license payments to continue working.

As an example from another domain, [5] mentions manufacturer and type codes printed on ICs and calls for a website to provide standardized information about well defined dependencies – much in the way that it is possible to find a datasheet describing an IC based on this type code. Such a website could be similar in concept to the OPC Foundation profile reporting website, but focus on functionality outside the area of OPC UA; the OPC UA address space could easily accommodate URIs pointing to “datasheets” about modules.

OPC UA leaves developers entirely free to handle dependencies which may be introduced by certain data presented in the address space. It also leaves them free to decide how to add information about such dependencies to the address space. However, it provides structures which can prove useful in this context, such as the advanced data modelling features already mentioned in the beginning of this section. For example, consider the case that a server (subsystem) is updated to expose a new widget. A client (subsystem) can immediately access the input and output data of this widget, but to use it efficiently, it must know how these inputs and outputs work together – the widget function. If the server adds type information to the widget object, the client can recognize it as the kind of meta-data which describes this function, and can decide whether to bring in complementary functionality on its side to make use of it. Still, the client will always remain in control of the choice whether to satisfy the dependency or not use the new widget.

VI. Adaptive flexibility in OPC UA

While the previous section discussed the role of OPC UA for de-coupling subsystems from the perspective of the server and client applications, this section focuses on the “internal mechanisms” of OPC UA. How do these mechanisms, which are usually hidden from the server or client application by the OPC UA SDK application programming interface (API), support adaptive flexibility? Again, the view taken is as far as possible a “black-box”, functional one, independent of how the mechanisms described in the OPC UA specifications are implemented.

A. Message passing

When (sub)systems are connected via OPC UA, all communication between them uses message passing following the request-response paradigm. Message passing requires that a request is stored in a separated (message) memory. Sender and receiver do not share a memory space; all passing of values has to be by copy, not reference. For this reason, message passing systems are also referred to as “shared nothing” systems. This greatly reduces the possibility for

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

undesired coupling to occur. Consider, for example, the situation that a software module is linked with another module which, by accident, contains a global variable of the same name, but different meaning or purpose. Such a name clash is impossible when these modules communicate via OPC UA, because the two variables will necessarily exist in separated memory spaces in this case. Moreover, the request-response message paradigm is naturally asynchronous in character. Unlike with a subroutine call within the same memory space, the module invoking the service always remains in control of its own program flow.

If implemented right, this basic principle ensures separation of states between the two subsystems and will block combinatorial effects from propagating between them. To this end, both the OPC UA middleware implementation (stack and SDK) and the application module which is using it for communication must react to these messages in a robust manner: most importantly, they must not count on the expected response to arrive promptly.

Let us consider this rule in further detail. First, the response may not arrive promptly, but with a significant delay, or it may not arrive at all. The subsystem must always take this possibility into account, it must not block unconditionally. Within OPC UA, timeouts for services and lifetimes for shared state between client and server (e.g., a logical channel or session) are specified. These timeouts can be negotiated between the communication partners to address varying response time requirements.

Second, if a response arrives, it may not be the expected one. It can also be an error message or an unknown response – unknown maybe because the communication partner was updated in the meantime. Such a situation must be handled as gracefully as possible. The subsystem must not hang or crash as a result.

Version transparency is a key quality in this context. If version transparency is implemented, response messages cannot be unexpected or unexpectedly absent – provided that the communication channel is reliable and that the entity on the other side is not malfunctioning. This is because with version transparency, it is the responsibility of the caller to only invoke the callee in ways that the callee can understand. In other words, in a scenario of controlled evolution, the caller must respect the version of the called entity. This is a logical consequence of the fact that when a fundamentally new requirement is added during the course of evolution of a system, older entities by nature will not have the capability to satisfy it. For example, a system designed to heat will not necessarily be able to cool as well, and a labeler for square boxes cannot label a round box.

However, an external system (such as one behind a connection element based on an OPC UA interface) may evolve in an uncontrolled way. It need not respect the version transparency theorems; the change in the external system may require our own system to adapt. This would cause a combinatorial effect. The connection element must block this combinatorial effect as far as possible, allowing us the choice of not adapting our own system; instead, we may want to display or log an error message saying the external system is not responding like it should – without causing our own system to crash or malfunction. The connection element must therefore catch unexpected responses to achieve version transparency. This is greatly facilitated by a mechanism which allows the older communication partner to declare that it does not understand or support a request, and give this response in a way that the younger can recognize it as such. For this purpose, OPC UA defines standardized response StatusCodes and gives the option to add free-form DiagnosticsInformation to further describe the reason of an error status.

B. *Technology mappings*

Already back in 1972, Dijkstra recognized difference in scale as a major source of our difficulties in programming [30]. Moreover, a widespread underestimation of the specific difficulties of size seems to be one of the major underlying causes of software failure.

An important design goal for OPC UA was scalability in terms of communication requirements. Small embedded devices should be allowed to contain a very basic OPC UA interface, while more powerful platforms (such as PC-based systems) should be able to provide more complex functionality. Considering their communication requirements, different levels in the automation pyramid are best served by different transport technologies. Communication on the level of ERP applications requires intermittent transport of large amounts of data at high data rates, while applications hosted in small embedded devices typically require continuous transport of small amounts of data with low latency (around 10 ms down to 10 μs).

OPC UA is defined independent of a particular network protocol and low-level encoding. It can be mapped to the most appropriate transport corresponding to the needs of an application. While this does not make it a full alternative to fieldbus systems (as no special provisions are made for time-deterministic communication), it certainly allows OPC UA to scale over a large range of requirements.

Being able to choose between transports does not only improve scalability. It also supports diversity: multiple transports may be able to fulfil a particular set of requirements. And with requirements changing and solutions improving over time, such an option to choose also introduces new possibilities with regard to the property of evolvability.

In 1994, DCOM was a good base technology for OPC. Over time, it became a limiting factor. DCOM was a vendor dependent technology and restricted the choice of operating system and programming language. Choosing a network protocol (and message encoding) as the basis already supports

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

multi-platform and cross-programming-environment communication better, since the concerns of sending a message to the communication partner and acting upon this message are separated. However, this still would not have allowed diversity with respect to transports. Therefore, a further step was taken: OPC UA was based on an abstract service oriented architecture. Part 4 of the specifications specifies the services in abstract terms, and Part 6 specifies the current transport technology mappings. The API is not standardized, giving the freedom to provide the modest appropriate solution for any programming language or framework.

The technology mappings do not only apply to communication protocols and message encodings, but also to security algorithms for encryption and authentication. By decoupling the specification of the OPC UA services from these implementation choices, as well as decoupling them from the programming environment, the “what” and “how” are very clearly separated. This manifestation of the separation of concerns principle is a very visible improvement in the process from classic OPC towards OPC UA.

Thanks to it being based on abstract service definitions, the stable core purpose of OPC UA – providing access to values and events in another subsystem – is decoupled from its more rapidly envolving technology environment. The choice to standardize a number of technology mappings stems from the necessity to prevent a completely fragmented landscape of implementations, which would not be interoperable with each other. Within these standardized technology mappings, profiles serve the purpose of selecting compatible, complementary subsets of diverse functionality. Finally, interoperability testing addresses the possibility of different interpretations of the specifications.

C. Further improvements over OPC Classic

In comparison with the classic OPC specification, several further improvements towards conformance with the principles of separation of concerns and version transparency can be found in OPC UA. Some of them are highlighted in the following.

First, one of the reasons why the authors of OPC UA used the word ‘unified’ in the name of the standard is that they unified previously different ways of accessing information. Classic OPC defined independent specifications, each covering a part of the functionality now provided by OPC UA. These specifications each had slightly different ways of doing the same thing, such as browsing the available data tags. By offering a unified way of access to this information, OPC UA provides increased reusability.

OPC UA clearly separates actions and objects. For example, OPC Classic defined a special GetStatus method to obtain status information about a server. In OPC UA, this information is modeled as the server status variable, is placed at a defined position in the server namespace, and can be accessed with the read service or monitored for changes just like any other variable node. This is clearly a manifestation of separation of concerns.

Layering is explicitly mentioned in [2] as a manifestation of the Separation of Concerns principle. In OPC UA, layering can be found in many places. For example, subscriptions (to notifications for changing values in the address space) are separated from an underlying session, and a session is separated from an underlying secure communication channel. These entities maintain their own state, which is independent of the state of the layers below them: subscriptions can be transferred between sessions (which supports smooth transfers between redundant clients or servers), and a session can switch over to another secure channel without being terminated (which enables transparent use of redundant network links).

Another example of layering can be found in the way how most OPC UA middleware is split from an implementation point of view. Basic protocol functions are covered by the OPC UA Stacks, which are made available by the OPC Foundation for its members. On top of such a Stack, commercial SDKs (Software Development Kits) are developed and marketed by OPC UA expert companies. Using these SDKs as a basis, developers create OPC UA client and/or server applications.

Finally, the current transport mappings for OPC UA contain a number of examples for action version transparency support. The OPC UA TCP Hello and Ack messages, which open every conversation between a client and a server using this protocol, contain ProtocolVersion parameters, and the server is required to accept newer protocol versions. The OPC UA Secure Conversation OpenSecureChannel service messages also include ClientProtocolVersion and ServerProtocolVersion parameters. If the web service mappings are used, SOAP as well as the WS-* standards encode the protocol version in every message by way of specifiying the XML namespace URI, which uniquely identifies the version. HTTP headers also contains a version field. Using this version information, the server can correctly interpret messages from an older client, or reject the request in a controlled manner if it has an unknown, more recent version. HTTP/1.1 [31] also specifies the Upgrade header, which a client can use to indicate to the server that it supports another version of the protocol (or other protocols in general) which it would like to use if the server finds this appropriate.

VII. Normalized modules within OPC UA middleware

As discussed in Section IV.B, we found it difficult to obtain cost estimates for implementing individual OPC UA profiles in an application program. Apparently, it is a problem to obtain useful estimates of the development effort, with or without the use of existing tools, in any programming language or development environment. Why could none of the experts, who we truly consider having acknowledged

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

excellence in their domain, provide concrete answers to our question?

The initial idea of the decision support tool included the assumption that it is possible to estimate the development effort or cost of implementing an individual OPC UA profile in a reliable way – without considering the specific context of such an effort. Estimating the cost of a software project is already a difficult task when context information (such as the current state of the software or the skills of the available developers) is known, but we sought a generic answer, independent of which other OPC UA profiles may possibly already exist in a considered implementation, and also independent of the size of the (existing) system.

This expectation is similar to what was expected from system analysts and software architects for over decades, i.e., listing up all the old, recent and future functional requirements of a system. Doing so seems to include very uncertain assumptions: in particular, the assumption that all dependencies of a newly added or changed feature are under control, are explicitly known, can be measured and estimated. The exponential character of the curve in Figure 5 illustrates that this assumption cannot be defended. The precise slope of this curve for a system is impossible to determine; therefore, it is impossible to give a precise estimate for any given system size. It is for this reason that the Normalized Systems Theory focuses on elementary, anticipated changes instead (as introduced in Section III-B).

It can be safely assumed that the experts' experience only concerned Lehman-type systems of some kind. In such a system, the development effort or cost of an OPC UA profile is not only a function of the functional requirements of the OPC UA profile itself, but also of the size of the (existing) system, including dependencies and potentially existing combinatorial effects. For a Normalized System, a reliable cost estimate would be much easier to give.

In general, the Normalized Systems Theory calls for radical separation of concerns and thus promotes a high granularity of modular systems. Even if this goal cannot be immediately attained, it will typically be a benefit to keep modules small in order to keep the impact of changes low. For purposes of illustration, the remainder of this section therefore explores how finely an implementation of OPC UA should be divided into modules to achieve separation of concerns. Given that the actual application (and all program logic to connect the OPC UA interface with it) is a separate concern for sure, we will focus on units of functionality defined in the OPC UA standard, which will usually be implemented by a Stack and/or SDK.

When looking at the OPC UA profiles specifications, we find four types of entities: profile categories, profiles, facets, and conformance units. Profile categories have no meaning in terms of software constructs, but only exist to structure the specifications for the purpose of readability. They provide major functional groupings, such as all profiles necessary to implement a complete server or client. Likewise, the distinction between facets and profiles has organizational purposes: a facet refers to a profile which is expected to make part of a larger profile. For the purposes of this discussion, facets and profiles can be treated the same.

Profiles describe features of applications. We have already identified profiles as a manifestation of the separation of concerns principle in Section V.C. In other words, when building an OPC UA application, it is possible to select and implement (or not) each individual profile. The features represented by OPC UA profiles are well separated and the profiles make explicit for a communication partner which features are supported and which are not. Profiles are the smallest unit of coherent functionality from the perspective of an OPC UA communication partner.

Profiles are comprised of ConformanceUnits. Conformance units are the foundation of the OPC compliance and certification program They are defined as a specific set of features that can be tested as a single entity. OPC UA Compliance testing activities include functionality testing, behavior testing, interoperability testing, load testing and performance testing. Conformance units are the smallest entities in the OPC UA standard from the perspective of compliance testing.

However, a ConformanceUnit can cover a group of services, portions of services or information models by definition. And, actually, services are the smallest entities which can be invoked by an OPC UA communication partner; with profiles as the smallest unit of coherent functionality from this perspective, they could be considered the smallest unit of individual functionality from the perspective of an OPC UA communication partner.

When examining how finely an implementation of OPC UA should be divided into modules to achieve separation of concerns, we must however take an "inside" instead of an "outside" view. Up to now, we have been taking a functional, or "black box" perspective. From this perspective, it is described what a module does, i.e., what its function is. Actually, functionality testing, which we mentioned in the context of ConformanceUnits, is also called black box testing: the test candidate is seen as one single black box – from the "outside". On the other hand, the constructional (or white box) perspective describes the subsystems of which a system consists, as well as the way in which these subsystems collaborate in order to bring forth the function as described in the black box perspective [29]. The Normalized Systems Theory contains rules with relation to the functional content of a module, the interaction between data and action entities, and the mutual interaction of action entities. In other words, it is concerned with the constructional perspective – the "inside" of a module.

If we consider an OPC UA service from the constructional perspective, i.e., from a developer's point of view, we realize that it requires some lower-level functionality. For example,

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

the network packet containing a service request or response has to be assembled (or disassembled), taking the correct message encoding into account. The basic encoding structure is the same for all services. For example, all messages contain a header identifying it as an OPC UA message. This common header is an example for a change driver as introduced in Section III.C. When it changes (for example, to reflect a new version of the OPC UA protocol), this affects all service implementations. It must therefore be reassigned to a separate module.

This example illustrates that it is a challenging goal to reach the high granularity required for Normalized Systems. A migration strategy focusing on selected interfaces and the encapsulation of external systems offers a transitional path.

VIII. Conclusion and outlook

Adaptive flexibility is an essential quality in modern information and communication systems. In our survey, we saw a great variety in requirements regarding industrial communication all over industry. While we were able to correlate OPC UA feature sets (i.e., profiles) with application requirements easily, the survey did not yield a conclusive set of profiles associated with any single field of application.

We consider OPC UA to be a highly valuable tool for supporting scalability, diversity and evolvability – that is, adaptive flexibility – in the domain of industrial communication and, in general, everywhere it can be used to de-couple subsystems with compatible communication requirements. When gauging OPC UA against the principles of Normalized Systems Theory, this becomes even more evident.

We find that the considerable tacit heuristic knowledge of the authors of OPC UA has lead to an amount of manifestations of several fundamental principles of Normalized Systems, such as separation of concerns. The user of OPC UA takes advantage of these manifestations without even being aware of them. OPC UA also does not prevent the separation of states between subsystems it connects.

For the same reason, some of the manifestations discussed may appear to be nothing but “established good programming practice” to seasoned software developers, who possess a significant amount of relevant tacit knowledge. The goal of NST is to make this knowledge explicit, and the discussion in this paper should also serve to illustrate the connection between theorems and established practice. In addition, it should be considered that embedded systems, including industrial automation systems such as PLCs, trail behind the forefront of software development to a varying, but significant extent: in some of these environments, name conflicts between global variables in modules are still an everyday problem. In this sense, the drive of advanced technologies such as OPC UA towards the embedded level can certainly be considered a welcome evolution. First SDKs for OPC UA on embedded systems are already commercially available, but remain a relevant target of ongoing work.

We also identified some features within the OPC UA specifications which can support adaptive flexibility and system-wide evolvability – both through their originally intended use as well as for applying NST concepts. According to personal communication, some of these features are still rarely used by practicioners. In particular, this seems to be true for the version attributes of the OPC UA objects. We feel that raising awareness of these features among practitioners, users and developers alike, could be beneficial for the industry.

After all, towards a truly Normalized System, the principles of the theory must also be followed in any application built on top of OPC UA. While OPC UA encourages applying these principles by way of its design, it does not enforce them. OPC UA stack, SDK and application programmers must still do their part to prevent the system from “hanging”.

Certainly, no stack, SDK or application program can be expected to become a fully Normalized System soon. Among other reasons, one faces the challenge of introducing new paradigms to practitioners when deploying Normalized Systems. Those who are not aware of NST theorems will find it harder to read and debug “normalized” code. However, an OPC UA layer can separate Lehman systems while being a Lehman system itself. OPC UA is an excellent foundation for connection elements, which again can be the base for a stepwise migration strategy towards Normalized Systems.

Due to the complexity of the OPC UA specifications, their evaluation in this single paper could not be complete. We believe there is ample room for future work to further investigate and elaborate them against the light of NST. Also, a deeper investigation of potential combinatorial effects could contribute to improving the quality of OPC UA applications.

Acknowledgment

This paper is part of the project IMA, funded by the Flemish Agency for Innovation by Science and Technology (IWT). The authors wish to thank all respondents of the survey and the OPC Foundation for their constructive collaboration. In particular, the authors would like to thank Paul Hunkar, one of the lead authors of the OPC UA profiles specification, who has provided valuable consulting expertise.

Acronyms

A&E	Alarms and Events
API	Application Programming Interface
BMS	Building Management System
CMD	Commands
DA	Data Access
DCOM	Distributed Component Object Model
DCS	Distributed Control System

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

EDDL	Electronic Device Description Language
ERP	Enterprise Resource Planning
FDI	Field Device Integration
HDA	Historical Data Access
HMI	Human Machine Interface
HTTP	Hyper Text Transfer Protocol
IC	Integrated Circtuit
IP	Internet Protocol
MES	Manufacturing Execution System
NST	Normalized Systems Theory
OPC	Open Productivity and Connectivity
PAC	Programmable Automation Controller
PLC	Programmable Logic Controller
SCADA	Supervisory Control and Data Acquisition
SDK	Software Development Kit
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
TFM	Technical Facility Management
UA	Unified Architecture
URI	Uniform Resource Identifier
WS	Web Service
XML	eXtensible Markup Language

References

[1] D. van der Linden, M. Reekmans, W. Kastner, and H. Peremans, "Towards agile role-based decision support for OPC UA profiles," in *Proc. 7th International Conference on Internet and Web Applications and Services (ICIW 2012)*, 2012, pp. 40–45.

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

[3] D. van der Linden and H. Mannaert, "In search of rules for evolvable and stateful run-time deployment of controllers in industrial automation systems," in *Proc. 7th International Conference on Systems (ICONS 2012)*, 2012, pp. 67–72.

[4] D. Van Nuffel, "Towards designing modular and evolvable business processes," Ph.D. dissertation, University of Antwerp, 2011.

[5] D. van der Linden, H. Mannaert, and P. De Bruyn, "Towards the explicitation of hidden dependencies in the module interface," in *Proc. 7th International Conference on Systems (ICONS 2012)*, 2012, pp. 73–78.

[6] D. van der Linden, H. Mannaert, and J. de Laet, "Towards evolvable control modules in an industrial production process," in *Proc. 6th International Conference on Internet and Web Applications and Services (ICIW 2011)*, 2011, pp. 112–117.

[7] D. van der Linden, H. Mannaert, W. Kastner, and H. Peremans, "Towards normalized connection elements in industrial automation," *International Journal On Advances in Internet Technology*, vol. 4, no. 3&4, pp. 133–146, 2011.

[8] D. van der Linden, G. Neugschwandtner, and H. Mannaert, "Industrial automation software: Using the web as a design guide," in *Proc. 7th International Conference on Internet and Web Applications and Services (ICIW 2012)*, pp. 67–72.

[9] *OPC Unified Architecture Specification*, OPC Foundation Std., Release 1.02, 2012–2013.

[10] *OPC unified architecture*, International Electrotechnical Commission Std. IEC 62 541, 2010–2012.

[11] R. Armstrong, "Implementing UA," OPC Unified Architecture Developers' Conference and Workshop 2008, Oct. 2008.

[12] J. Luth, "OPC UA Architecture," OPC Unified Architecture Developers' Conference and Workshop 2008, Oct. 2008.

[13] T. Hannelius, M. Salmenpera, and S. Kuikka, "Roadmap to adopting OPC UA," in *Proc. 6th IEEE International Conference on Industrial Informatics (INDIN 2008)*, 2008, pp. 756–761.

[14] PLCopen and OPC Foundation, *OPC UA Information Model for IEC 61131-3, Release 1.00*, 2010.

[15] D. Grossmann, K. Bender, and B. Danzer, "OPC UA based field device integration," in *Proc. SICE Annual Conference*, 2008, pp. 933–938.

[16] OPC UA ISA-95 Working Group, *OPC UA for ISA-95 Common Object Model, Release Candidate 1.01.00*, 2013.

[17] "OPC UA Profiles," http://www.opcfoundation.org/profilereporting/, last accessed June 2013.

[18] 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*, vol. 76, no. 12, pp. 1210–1222, 2011.

[19] D. Parnas, "Software aging," in *Proc. 16th International Conference on Software Engineering (ICSE-16)*, 1994, pp. 279–287.

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

[21] H. Mannaert, J. Verelst, and K. Ven, "Towards evolvable software architectures based on systems theoretic stability," *Software: Practice and Experience*, vol. 42, no. 1, pp. 89–116, 2012.

[22] E. W. Dijkstra, "On the role of scientific thought," in *Selected writings on computing: a personal perspective*. Springer, 1982, pp. 60–66.

[23] I. Yoon, A. Sussman, A. Memon, and A. Porter, "Testing component compatibility in evolving configurations," *Information and Software Technology*, vol. 55, no. 2, pp. 445–458, 2013.

[24] C. Y. Baldwin and K. B. Clark, *Design rules: the power of modularity*. MIT Press, 2000.

[25] S. Sharma, *Applied Multivariate Techniques*. John Wiley & Sons, 1996.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[26] W. Mahnke, S. H. Leitner, and M. Damm, *OPC Unified Architecture*. Springer, 2009.

[27] J. Lange, F. Iwanitz, and T. Burke, *OPC – From Data Access to Unified Architecture*. VDE-Verlag, 2010.

[28] D. Campagnolo and A. Camuffo, "The concept of modularity within the management studies: a literature review," *International Journal of Management Reviews*, vol. 12, no. 3, pp. 259–283, 2009.

[29] P. De Bruyn and H. Mannaert, "Towards applying Normalized Systems concepts to modularity and the systems engineering process," in *Proc. 7th International Conference on Systems (ICONS 2012)*, 2012, pp. 59–66.

[30] E. W. Dijkstra, "Notes on structured programming," in *Structured programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. Academic Press Ltd., 1972, pp. 1–82.

[31] IETF, "Hypertext transfer protocol – HTTP/1.1," Jun. 1999, RFC 2616.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org