# Towards evolvable state machines for automation systems

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

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

***Abstract*—Since several decades, the pressure on organizations to swiftly adapt to their environment has been increasing. This also applies to industry. One of the consequences is the increasing importance of evolvability in production control systems. Many such systems are modelled using finite state machines. This paper presents an explorative attempt to define design rules and constraints that should be applied to state machines to enable evolvability. Our design of an evolvable state machine is based on normalized systems theory. Without loss of generality, this design is inspired by the typical requirements of automation systems.**

***Keywords*-Normalized Systems; Evolvability; Finite State Machines; Automation Systems.**

## I. INTRODUCTION

Current organizations need to be able to cope with increasing change and increasing complexity in most of their aspects and dimensions [1]. We shall call a system *evolving* when changes in terms of the system's capabilities occur. The effort or cost required for adding or changing a specific capability is a property of a system – the property of *evolvability*. Evolvability is increasingly important for organizations to allow them to swiftly adapt to an agile and complex environment. This paper focuses on the evolvability of production control and automation systems.

In general, the behaviour of any system, subsystem or process can be categorized as *static* or *dynamic*. Static behaviour means that the (sub)system does not experience any internal change during its lifetime. A system is said to have dynamic behaviour if it changes its behaviour during its existence [2]. If dynamic reconfiguration can be achieved, downtimes of production control and automation systems can be reduced: a change which can be performed without a complete shutdown is called a 'dynamic reconfiguration', while a 'static reconfiguration' requires the complete shutdown of a system [3].

Efforts to improve the flexibility and maintainability of automation systems go back decades. The first approach to implement automation control logic was based on hard-wired relay systems. In the late 1960s, GM Hydramatic issued a request for proposal for an electronic replacement. The result was a Programmable Logic Controller (PLC), built by Bedford Associates. One of the main advantages was that changes in control logic could now be made by changing the program rather than changing wiring and bypassing or adding relays. In addition, programs could be reused for another application. The technology shift from hardware to software provided more flexibility and an improvement of maintainability. While this was certainly an improvement, the characteristic of evolvability is still not totally reached. Changing or debugging a few lines of software code is easier than rewiring relay circuits, but the number of software problems and bugs does not grow proportionally with the software size. Instead, they grow out of proportion. After reaching a certain size, software becomes a problem in its own right [4].

Besides maintainability, reusability and flexibility, evolvability is a critical non-functional requirement on software. In their review of evolvability as a characteristic of software architectures, Ciraci and van den Broek [5] define it as "a system's ability to survive changes in its environment, requirements and implementation technologies." However, evolvability is hard to measure, and existing software development methodologies focus on functional requirements almost exclusively.

In order to better satisfy non-functional requirements, giving software a clear, well-defined and if possible standardized structure is essential. This includes the use of standardized programming languages. As further discussed in Section IV of this paper, PLCs are usually programmed in one of the languages of the IEC 61131-3 standard [6]. Another important technique is to employ formal models wherever possible. A prime example in this context are finite state machines. For example, the ISA-88 standard recommends specifying elementary operations in batch manufacturing processes (e.g., filling a tank) by way of state machines. In the simplest case, the state machine for a so-called "equipment phase" contains the states "Idle", "Running" and "Complete". An equipment phase specification will, among other things, describe additional states, the conditions for transitions between these states (e.g., after a specified amount of time has elapsed) and the actions to take upon such a transition (e.g., close a valve).

While state machines are a valuable tool to increase system maintainability, they do not automatically guarantee

 ISBN: 978-1-61208-246-2 

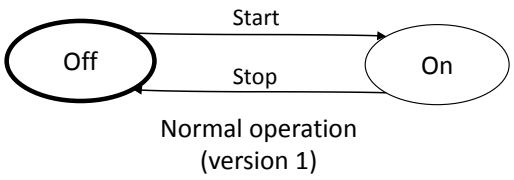


Figure 1. Motor control state machine, version 1

evolvability. The research presented in this paper focuses on the design of an evolvable state machine which can be implemented in one or more of the IEC 61131-3 languages, independent of vendor or CPU type. The design supports dynamic reconfiguration wherever possible.

The remainder of the paper is structured as follows: Section II gives an example of a state machine in an automation system going through subsequent evolution steps. Section III explains the basics of the normalized systems theory, which offers a formal guideline to system evolvability. Section IV introduces a concept for cross-vendor PLC system evolution which supports the coexistence of different versions of the same reusable, generic module. Section V proposes a set of design rules for evolvable state machines in the context of the concept presented in Section IV, derived from normalized systems theorems. In addition, implementation considerations are discussed. Section VI concludes the paper.

## II. An Evolving State Machine

A finite state machine is an abstract machine that can be in one of a finite number of states. The machine is in only one state at a time; the state it is in at any given time is called the current state. The current state can change upon a triggering event or condition. This is called a transition. A particular state machine is defined by the list of its states, the possible transitions between them, and the triggering condition for each transition.

In automated production installations, the behaviour of a large portion of the control equipment (such as valves, light curtains, pumps, mixers or conveyers) can be – and is – modelled using state machines. In the following, a motor control state machine is presented as an example. As the automation system evolves in response to new requirements and hardware capabilities, new versions of the state machine are introduced.

The initial version of the state machine, shown in Figure 1, shows a very simple model. The motor has two states, 'On' and 'Off'. Two conditions affect the state of the motor: the 'Start' and the 'Stop' command.

This is a very idealized and simplistic view. In an actual industrial environment, additional conditions such as failure conditions or interlocks must be taken into account. As an example for such an additional condition, the second version of the state machine considers the condition of a fuse.

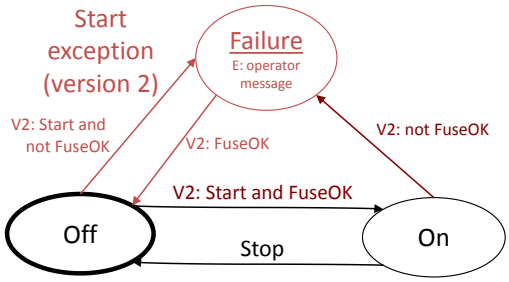


Figure 2. Motor control state machine, version 2

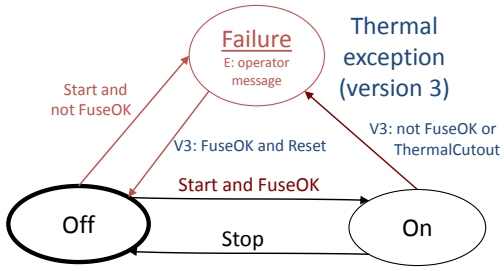


Figure 3. Motor control state machine, version 3

The start condition becomes 'Start and FuseOK' (Figure 2). The additional state 'Failure' is introduced. The condition for transitioning from the 'Off' state to the 'Failure' state is 'Start and not FuseOK'. The condition to go from the 'Failure' state to the 'Off' state is 'FuseOK'. If the fuse blows during the 'On' state, a transition to the 'Failure' state results. In addition, when entering the failure state, a notification is made to trigger an operator to solve the issue.

The third version considers the situation that the motor can stop due to a thermal cut out in the 'On' state (Figure 3). This is modeled with a new version of the transition to the 'Failure' state. In addition, the operator must push a reset button before the 'Off' state can be entered again after a failure, which is reflected by a new version of this transition as well.

## III. Normalized Systems Theory

Software undergoes an aging process, as recognized by Parnas [7]. Since there are indications that this aging process is also happening with business processes [8], we must consider the possibility that this phenomenon may actually apply to all non-physical systems in general which undergo an evolution in our society and economy.

For software systems, Manny Lehman formulated the law of increasing complexity, expressing the degradation of a system's structure over time [9]:

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

 ISBN: 978-1-61208-246-2

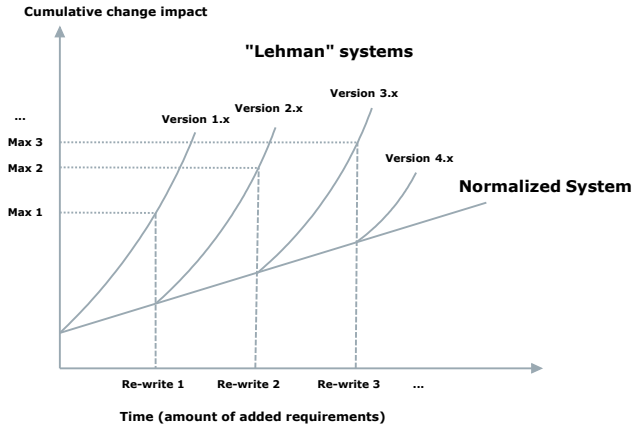


Figure 4. Improving software structure with a re-write [10]

In software development, this deterioration progresses with each update or hotfix. Over time, the deficiency of the structure renders the system unworkable. To mitigate this problem, a re-write of the whole system can help (Figure 4 [10]). The theory of normalized systems was introduced to challenge Lehman's law [11]. Other than most previous efforts to achieve maintainability of software, the contribution of the normalized systems theory goes beyond heuristics; instead of only advocating guidelines such as "low coupling and high cohesion", it provides theorems to derive yes/no answers to questions about evolvability.

In the context of normalized systems, an action entity shall be defined as a module which contains functionality, and a data entity shall be defined as a set of tags (fields). Action entities and data entities are the two main elements from which a system can be constructed. Action entities use data entities as input and output parameters. States, conditions, commands or events can be stored in a data entity. The four core theorems of normalized systems are:

1) Separation of concerns: *An action entity can only contain a single task.*
   A task is functionality which can evolve independently. If the system's developer anticipates that two or more parts of the core functionality can change independently, these parts must be separated. Therefore, normalized systems shall be constructed of action entities dedicated to one core activity.
2) Data version transparency: *Data entities that are received as input or produced as output by action entities must exhibit version transparency.*
   It must be possible to update one or more data entities which are passed between action entities and let multiple versions co-exist without affecting other versions of action entities.
3) Action version transparency: *Action entities that are called by other action entities must exhibit version transparency.*
   It must be possible to update an action entity, which is coupled with another action entity, while multiple versions of both modules can co-exist. In other words, introducing a new version of an action entity shall not require changes to any other action entity.
4) Separation of states: *The calling of an action entity by another action entity must exhibit state keeping.*
   Every action entity must keep track of its requests to other action entities. If the response to a request is not as expected, the calling action entity must not block indefinitely; rather, it shall handle the exceptional situation as appropriate for its own state.

Two additional theorems have recently been introduced as extensions of the theorems on data and action version transparency [12]. They address the challenge of managing the diversity of run-time *instances* of data and action entities in an evolving system.

5) Data instance transparency: *A data instance has to keep its own instance ID and the version ID on which it is based or constructed.*
   If the type definition (source code) of a data entity is updated to a new version, instances based on the previous version continue to exist in the system.If an action entity receives a data instance for processing, the action entity must have a way of knowing the version of this data instance to be able to handle the data instance in a version-compliant way. Therefore, every data instance must contain a version ID reflecting the version of the data entity it is an instance of. The instance ID serves to tell apart multiple instances of the same version.
6) Action instance transparency: *An action instance has to keep its own instance ID and the version ID on which it is based or constructed.*
   When run-time instances of action entities interact, they must consider the fact that the other action entity can be based on one of various versions of its type definition. A version ID is necessary to give the calling action instance information about which interactions are possible. Again, the instance ID serves to tell apart multiple instances of the same version.

Note that there is no supersedure relationship between *versions* in our context; versions may continue to exist in parallel indefinitely to reflect variants of a type of equipment.

## IV. Cross-vendor programs and instance version diversity in a PLC system

When an update of a PLC project includes the introduction of a new CPU type, or even a conversion of the software to another brand, software developers often re-engineer the whole project. Also, when a motor is replaced

 ISBN: 978-1-61208-246-2 

with a different one – or the update includes the introduction of a frequency drive, which requires a vendor dependent system function block –, engineers tend to re-write the module which is controlling the motor.

In the following, a concept is introduced to reduce the amount of these re-writes. A new motor should no longer require a new software module; neither should the entire project require re-engineering because of changing to a new brand or PLC family. To achieve this, we propose that programming is based on generic modules. There shall only be one module for every core function (e.g., motor control), with the variations between physical motors and their control being addressed by multiple, co-existing, versions of this module.

In this concept, the vendor independence brought about by the IEC 61131-3 standard is an important element. In the first decades after the introduction of the first PLC, these controllers were programmed in vendor-dependent languages. However, during the nineties, the need for a more generalized way to program PLCs increased. As a result, the IEC 61131-3 standard was introduced. It was definitely successful: nowadays, most common brands support at least some of the IEC 61131-3 languages. However, the standard does not include hardware configuration. Consequently, the connection to process hardware (process I/O) remains vendor-dependent. In addition, the standard allows some liberties (e.g., implementation-dependent parameters in Annex D [6]). Commercial IEC 61131-3 programming environments show some differences. Therefore, developers still often re-write a whole software project in case another brand of PLC is required.

Our approach to truly generic, vendor-independent PLC programming is shown by way of an example in Figure 5. A generic module, which strictly sticks to IEC 61131-3 code, contains the core functionality of a device, for example controlling a motor. Instances of this module represent individual motors. Before the generic module can be downloaded to a specific brand of PLC in order to control a specific motor, it undergoes an automatic vendor-mapping procedure, which converts part of the module code according to what is required by the vendor's specific environment. In addition, the vendor-mapping procedure adds an extra module to the (mapped) core module: a connection entity (CE). This connection entity is dedicated for a specific motor (instance), and includes all the details needed to connect the (mapped) core functionality with the process hardware (I/O). If necessary, this connection entity can also include vendor-specific function blocks (e.g., a scaling block for analog values, or a system block dedicated to control a specific frequency drive).

Each new version of the functionality is referred to as a class version, and each individual physical motor as an instance. Class versions correspond to the functionality *available* in the PLC (potentially in several co-existing versions), while instance versions correspond to the *instantiated* functionality for controlling a specific type of motor. Instance IDs refer to one single, specific physical motor; they tell the connection entity which hardware addresses an individual motor control module instance has to be connected to.

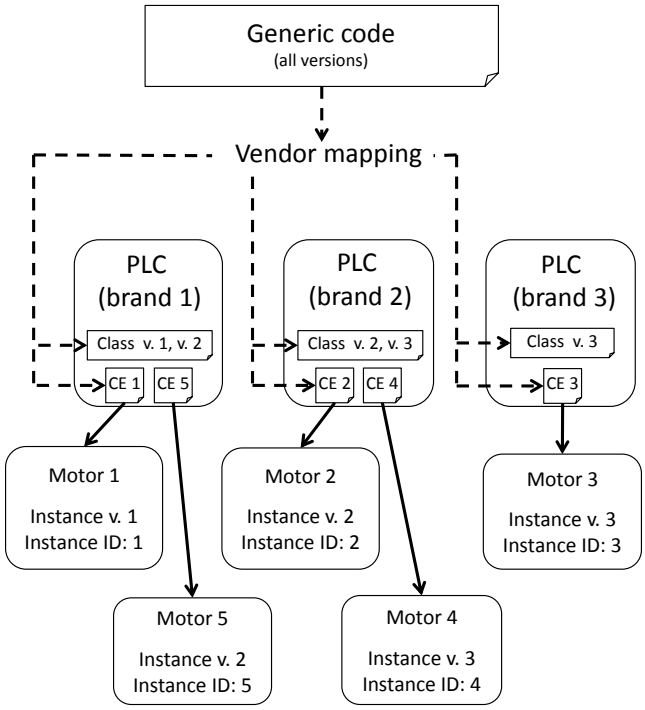


Figure 5. A generic software module with heterogeneous instances

For example, motor 1 could be controlled by an instance of version 1 of the state machine presented in Section II, since status feedback is not required for it. Motor 2 is controlled by an instance of version 2 of this state machine, since for it the fuse condition must be taken into account. Motor 5 is also controlled by an instance of version 2 of the state machine; while motor 2 and motor 5 thus share the same instance version, they have different instance IDs.

## V. EVOLVABLE STATE MACHINES

The previous sections discussed the benefits of version transparency and co-existence. In the following, we propose rules – based on normalized systems theory – that shall be followed by state machines and the program code implementing them in order to achieve these properties, and, thus, evolvability.

S1. *The functionality of a state machine shall be implemented in an action entity, while the state and transition trigger information should be stored in a separate entity – a data entity.*

When the functionality of a state machine is updated, a new class version is introduced. We want to be able to

 ISBN: 978-1-61208-246-2 

deploy instances of this new version to the system without disrupting the operation of instances of previous versions, which may still be adequate for part of the equipment. Thus, several instances of devices controlled by different versions of the state machine should be able to co-exist, and we require the logic of the state machine to change independently of these instances. Remember that if two parts of a module can change independently, they shall be separated following the separation of concerns principle (Theorem 1); from this follows the separation called for in this rule.

Every version of the data entity contains state and condition fields. In each state, a particular state of the associated process hardware is effected (e.g., letting the motor run in the 'On' state). This is done by way of a connection entity. Values for the condition fields are provided by other action entities (in particular, connection entities); when the conditions for a particular transition are fulfilled, the state machine action entity changes the current state of the instance.

S2. *The state machine data entity shall include an instance ID.*

The instance ID allows the connection entities to map this instance to the correct hardware addresses. This is necessary so that changes on hardware inputs are reflected as changes in transition fields, which in turn will cause the state machine action entity to perform the appropriate state transition. Likewise, the mapping is necessary in order for the required hardware outputs to be set to perform the action associated with a state of the state machine.

For example, in version 1 of the state machine presented in Section II (Figure 1), the output should be 'TRUE' when in the 'On' state and 'FALSE' in the 'Off' state.

S3. *The state machine action entity shall include a class version ID, and the state machine data entity shall include an instance (data type) version ID.*

To comply with the version transparency theorems, the data entity must contain its own version (the version of the state machine it is an instance of). This version ID lets action entities recognize the class version corresponding to the instance and act accordingly. The action entity should store its class version on the moment of compilation as a hard-coded constant.

Following our first rule (S1), the data and the functionality within the system should be separated. Therefore, we have a data entity to store the system's data in one or more data fields, and an action entity to perform actions based on the data in this data entity. Several versions of both data entities and action entities have to be able to co-exist. When a recent action entity instance encounters an older data entity instance, it must interpret its data fields in the way the older action entity instance would have. If necessary, default values need to be defined for fields not present in the older data entity instance. When a more recent data entity instance is processed by an older action entity instance, only old data fields are used, because the older action entity instance is not aware of the recently added data field(s). To enable proper interaction with instances of older versions, or at least prevent version conflicts, instance version IDs are required (Theorems 5 and 6).

For example, suppose we have an action entity version 2 (class version), which should process a data entity instance version 1 (data type). After reading the data entity instance's version ID, the action entity decides to never manipulate the 'FuseOK' data field, nor allows any transition to the state 'Failure'. These actions must be prevented because these fields do not exist in version 1 of the data entity; undefined behaviour would result. Instead, any information on the fuse or thermal cut out (if available) is ignored, corresponding to the (older) functionality of action entity version 1. Conversely, consider an instance of action entity version 1, which should process a younger instance of data entity version 2. This action entity instance is not even aware of the existence of fuse information nor the state failure, so it will never read nor manipulate these fields.

The potential 'ThermalCutout' transition of version 3 from the 'On' state to the 'Failure' state will simply never happen if not both the data entity instance and action entity are of version 3. In addition, the action entity must include a selection to decide whether or not a reset command from the operator is needed for the transition from the 'Failure' state to the 'Off' state (Figure 3).

S4. *States or transitions shall not be deleted between versions.*

The version transparency theorems (Theorems 1 and 2) state that multiple versions of both data entities and action entities have to be able to co-exist. If a state or a transition is deleted during an update of a state machine, the behaviour of instances of older versions of the action entity can become undefined. If following the transition logic of an old version a state should become active which does not exist any more after an update, the older version cannot co-exist with the recent one. Consequently, deletions of states or transitions are violations of the version transparency theorems.

S5. *State modifications shall apply transparent coding or version wrapping.*

Deletions of states can cause violations of the version transparency theorems, so only additions and modifications are allowed. Modifications shall adhere to the principles of transparent coding or wrapping versions [13]. Transparent coding is defined as the writing of internal code in a module which is not affecting the functionality of previous versions. When transparent coding is not possible (e.g., because of conflicting functionality of the versions, or when the combination of the functionality of different versions requires too complex code), version wrapping can be applied. Following this principle, different versions of a module co-exist in

 ISBN: 978-1-61208-246-2 

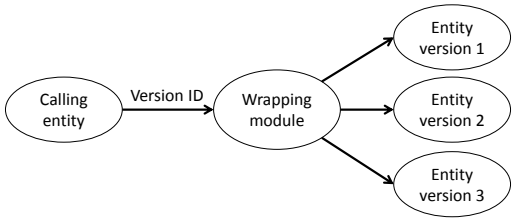


Figure 6. The concept of version wrapping

parallel, and a wrapping module selects the desired version based on the version ID (see Figure 6).

*Summary*

In summary, in order to attain the property of evolvability for a state machine, updates must be confined to the following set of anticipated changes:

- An additional state
- An additional transition
- A new version of a state following the principle of transparent coding
- A new version of a transition following the principle of transparent coding
- A new version of a state following the principle of version wrapping
- A new version of a transition following the principle of version wrapping

## VI. Conclusion

The state machine is a valuable artefact for modelling systems. However, when systems evolve, it follows from Manny Lehman's law of increasing complexity that their further evolution is restrained when the systems' size increases over time. In a rapidly changing environment, there is a need for evolvable state machines: when production systems evolve, corresponding changes have to be made in the automation software.

This paper presented a design for evolvable state machines that can be used in automation systems software. The design is based on the normalized systems theory. Rules were derived to constrain changes to state machines in order to achieve the property of evolvability. In addition, case scenarios were discussed showing how instances of different versions of such an evolvable state machine can coexist.

The design supports dynamic reconfiguration, as called for by Kuhl and Fay, to update a system without the need for a complete system shutdown. Compiling an IEC 61131-3 project includes allocating memory to variables. A shutdown is only necessary when this memory must be remapped. Changing the value of a data field in a data instance can be done without recompilation, so no shutdown is required. When, for example, a motor is replaced by a new one, this change is reflected by a change to the instance version ID of the data entity in our design. Therefore, dynamic reconfiguration is supported for such a situation.

Regarding future work, state machine libraries and toolkits should be improved by adding constraints to follow the rules presented in this paper, increasing system evolvability by ensuring compliance with the theorems on normalized systems.

## References

[1] H. Mannaert, P. De Bruyn, and J. Verelst, "Exploring entropy in software systems: Towards a precise definition and design rules," in *Proc. $7^{th}$ Intl. Conference on Systems (ICONS 2012)*, 2012, pp. 93–99.

[2] J.-P. Kruth, T. V. Ginderachter, P. I. Tanaya, and P. Valckenaers, "The use of finite state machines for task-based machine tool control," *Computers in Industry*, vol. 46, no. 3, pp. 247–258, 2001.

[3] I. Kuhl and A. Fay, "A middleware for software evolution of automation software," *Proc. $16^{th}$ IEEE Intl. Conf. on Emerging Technologies and Factory Automation (ETFA)*, 2011.

[4] O. Niggemann, "System-level design and simulation of automation systems," in *Proc. $8^{th}$ IEEE Intl. Workshop on Factory Communication Systems (WFCS)*, 2010, pp. 173–176.

[5] S. Ciraci and P. van den Broek, "Evolvability as a quality attribute of software architectures," in *Proc. $2^{nd}$ Intl. ERCIM Workshop on Software Evolution*, 2006, pp. 29–31.

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

[7] D. L. Parnas, "Software aging," in *Proc. $16^{th}$ Intl. Conf. on Software Engineering (ICSE)*, May 1994, pp. 279–287.

[8] D. Van Nuffel, "Towards Designing Modular and Evolvable Business Processes," Ph.D. dissertation, University of Antwerp, 2011.

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

[10] D. van der Linden, G. Neugschwandtner, and H. Mannaert, "Industrial automation software: Using the Web as a design guide," in *Proc. $7^{th}$ Intl. Conf. on Internet and Web Applications and Services (ICIW)*, 2012.

[11] 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.

[12] 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. $7^{th}$ Intl. Conf. on Systems (ICONS)*, 2012, pp. 67–72.

[13] 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.

 ISBN: 978-1-61208-246-2 