In Search of Rules for Evolvable and Stateful run-time Deployment of Controllers in Industrial Automation Systems

Dirk van der Linden
Electro Mechanics Research Group
Artesis University College of Antwerp
Antwerp, Belgium
dirk.vanderlinden@artesis.be

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

***Abstract*—Automation systems in the domains of smart grids, digital factories and modern process systems struggle to follow the permanent shift of their requirements. Hence, the most prominent non-functional requirement of a system seems to be evolvability. The recently proposed Normalized Systems theory has formulated constraints on the modular structure of software architecture in order to engineer evolvable systems. In this context, evolvability is related to systems theory stability as it is defined as the possibility to perform additional anticipated changes to the system of which the output remains bounded, even if an unlimited systems evolution is assumed. In this analysis, one considered the context of compile-time. However, this view becomes far more complex during run-time deployment, because some modules have several instances, others only one. The amount and complexity of connections during run-time is not straightforward visualizeable. In this paper, we introduce two new theorems, which are complementary with the existing four, to achieve a stateful run-time deployment.**

***Keywords*-Normalized Systems; Evolvability; Systems Theory; Modularity; Industrial Automation.**

I. INTRODUCTION

The non-functional requirement of evolvability is a very desirable characteristic for both information and production control systems. First, current information systems still struggle to provide high levels of evolvability [1]. Indeed, software maintenance is regarded as one of the most expensive phases of the software life cycle, and often leads to an increase of architectural complexity and a decrease of software quality [2]. However, contemporary organizations are increasingly faced with changing environments, which emphasizes the need for evolvability of software systems. The widely accepted shortage in programming manpower, and the disappointing success rate in business software development projects, call for a major gain in this kind of software development. Second, automation software should be able to evolve over time as well. This is a key requirement in the beginning age of decentralized energy generators and consumers prominently known as smart grid [3]. The upcoming of PLCs (Programmable Logical Controllers) some 40 years ago, has provided more flexibility to develop and maintain automation systems in terms of software in spite of hardware (i.e., electrical circuits). The dynamic interchange of software components of a PLC with near-to-zero downtime some years ago, has provided the flexibility to alter automation systems while staying in full service [4]. The modification of an automation system should be possible without affecting existing parts, even if running parts are reused in a so-called online change (i.e., downloading a new software part to the controller without stopping the system).

Normalized Systems theory has recently been proposed to contribute in achieving the characteristic of evolvability in systems. Requiring stability as defined in systems theory, four design principles or theorems are proposed. Systems built with modules, which comply with these theorems, can increase without losing control over the so-called combinatorial effects of a change. A bounded set of anticipated changes should result in a bounded amount of impacts to the (growing) system.

One can visualize an overview of a system by placing a number of modules on a surface, and connecting them through their interfaces. Since a good interface is roughly explaining the core functionality of a module, it may seem rather straightforward to consider the relations between the modules via their connections. However, from the moment the (compiled) code starts to run, obtaining this overview is even far more complex. For instance, some modules have in run-time several instances, others only one. The amount, complexity and dimensions of connections during run-time is not straightforwardly visualizeable. However, obtaining such overview of the runtime situation in an automation project is necessary to control evolvability, and predict combinatorial effects. There is need to minimize downtimes by dynamic reconfiguration of a system, without a complete shutdown. It is important to note the contrast with a static configuration, which does need a complete shutdown of the system. Such a static reconfiguration is very costly and should be replaced by a dynamic one [4].

In this paper, we introduce two new theorems, which are complementary with the existing four, to achieve a stateful and an evolvable run-time deployment.

The paper is structured as follows. In Section II, we

 ISBN: 978-1-61208-184-7

mention some related work. In Section III, the Normalized Systems theory will be discussed. In Section IV, we introduce the two new theorems. Finally, conclusions and future research are discussed in Section V.

II. Related Work

One of the motivations of Dijkstra to argue for the abolishment of the GOTO statement from all "higher level" programming languages was the finding that "...our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible." [8]

Most modern higher level programming languages do not allow GOTO statements, but we think more rules can contribute to address Dijkstra's ambition. Even without the GOTO statement the "dynamic process" (i.e., its dispersal in time) is still difficult to overview. Designing and developing modules, connected through interfaces with other modules, may seem rather straightforward when they are modeled in a static way (i.e., dispersed in text space or graphical visualizations). Studying the behavior of these modules in a dynamic view is even more complex. For example, one type or class definition results in run-time deployment in several instances. Some modules have a lot of instances, others only one. Consequently, representing the behavior of modules, together with their interrelations and instances during run-time is far from easy.

Additionally, visualizing and overviewing processes is harder if the system becomes more complex. Reducing complexity is a way to facilitate the formation of overviews. Employing meta data and information modeling contributes on this field. For example, Mahnke et al. have proposed classifications of types of information modeling standards for automation [9]. Also, they discuss the applicability of possible approaches to expose those models.

Further, Kuhl and Fay introduced an approach to modify automation systems by way of a middleware concept [4]. They focus on reconfiguration of systems, even during run-time, when a shutdown is not possible. This reconfiguration should not affect existing, running parts. Consequently, in such a system we have running instances, which are instantiated with a specific type version, and new constructions, which provide instances with a new type version. These instances should be able to co-exist, even when they are slightly different because of the different type versions they are based on.

Version transparency is one of the key points of the Normalized System theory to achieve stability [10]. As a consequence, different versions of both data entities and action entities can co-exist simultaneous. At first sight, the co-existency of different versions is not contributing in making the correspondence between the program and the run-time deployment trivial (i.e., what Dijkstra called for). However, allowing only one version (typically the most recent one) in an evolving system leads to unbounded combinatorial effects. The principle of version transparency is providing the possibility to overview nevertheless the different versions. The question is, how can we achieve an overview of run-time instances of these primitives, each constructed in one of the co-existing versions.

III. Normalized Systems

The law of Increasing Complexity (Lehman [11]) states: "As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it". This degradation of a system's structure over time is well known. More difficult to determine is the detailed cause of this deterioration. Which new parts of the system contribute in the effects of this law? In other words, why is a piece of code causing more costs in the *mature* stage of the lifecycle of a system, than exactly the same piece of code, is causing in the *beginning* stage of the project? The authors of the Normalized Systems theory combine Lehman's law of Increasing Complexity with the assumption of unlimited systems evolution: *The system evolves for an infinite amount of time, and consequently the total number of requirements and their dependencies will become unbounded*. These authors admit that in practice this assumption is an overstatement for the most commercial applications, but it provides a theoretic view on the evolvability issue, which is independent of time. The rather vague questions like "Is this change causing more troubles than another?" can be replaced by the fundamental question: "Is this change causing an unbounded effect?". The authors of Normalized Systems want to provide a deterministic and unambiguous yes/no answer on this question, by evaluation whether one of the theorems is violated or not.

A. Stability

The single postulate, from which the Normalized Systems theory is derived from, states that *a system needs to be stable with respect to a defined set of anticipated changes*. In systems theory, one of the most fundamental properties of a system is its stability: a bounded input function results in bounded output values, even for $T \rightarrow \infty$ (with T representing time).

Stability demands that the impact of a change only depends on the nature of the change itself. Conversely, changes causing impacts that are dependent on the size (or amount of changed or added requirements) of the system, are called *combinatorial* effects. To achieve stability, combinatorial effects should be abolished from the system. Systems that exhibit stability are defined as *Normalized Systems*. Stability

 ISBN: 978-1-61208-184-7

can be seen as the requirement of a linear relation between the cumulative changes and the growing size of the system over time. Combinatorial effects or instabilities cause this relation to become exponential (Figure 1). By eliminating combinatorial effects, this relation can be kept linear for an unlimited period of time, and an unlimited amount of (anticipated) changes to the system.

B. Design Theorems for Normalized Systems

Anticipating all the desired changes of the future might seem as a rather daunting task. Indeed, lots of system analysts get lost in this ambition. The authors of Normalized System's theory do not state they can do better in listing up all the functional requirements, possibly hidden in the present, or desired in the future. Fulfilling this task would be very complex and exceptional. These authors want to introduce another approach to achieve the same goal. The discussion about *anticipated changes* is not about changes, which are directly associated to recently expressed desires of the customers or managers to improve a system. Instead, anticipated changes focus on *elementary* changes, associated to software primitives. Typically, one real-life change corresponds with a lot of elementary changes, expressed in terms of software primitives. The Normalized System's theory is not focussing on the real-life changes, but on the elementary changes. In this section, we give an overview of the design theorems or principles of Normalized Systems, i.e., systems that are stable with respect to a defined set of anticipated (elementary) changes:

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

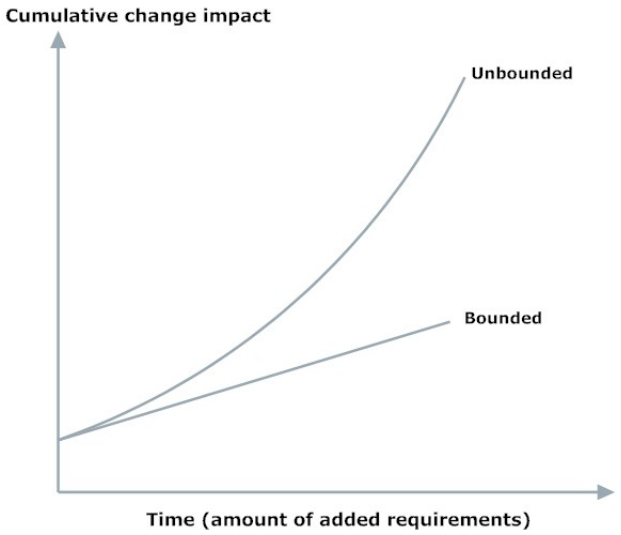

Figure 1. Cumulative impact over time

These changes are associated with software primitives in their most elementary form. Changes to meet "high-level requirements" that are obtained by system analysts from traditional gathering techniques (including interviews and use cases) [12] should be converted to these abstract, elementary anticipated changes. We were able to convert all high level changes in several case studies to one or more of these abstract anticipated changes [1][13][14]. However, some issues we encountered during the implementation of these proof of principles [14], have led to the introduction of the two new theorems of the next section. The systematic transformation of real-life requirements to the elementary anticipated changes is outside the scope of this paper. Note that already initial efforts are done in mapping the organizational requirements to the primitives of Normalized Systems [15].

1) Separation of concerns:

An action entity can only contain a single task in Normalized Systems.

This principle is focussing on how tasks are implemented within processing functions. Every *concern* or task has to be separated from other concerns. The identification of a task should be based on the concept of change drivers. A task is something that is subject to an independent change. A single change driver corresponds to a single concern in the application.

Proof: Consider a module M containing a task A and a second task B. The evolution of task B causes the introduction of N versions of task B. Since task B is part of module M, module M has also N versions. The introduction of a mandatory version upgrade of the task A will require to upgrade all N versions. According to the assumption of unlimited systems evolution, N will increase over time and will become unbounded, and so will the number of versions of task B. As a result, the number of additional version upgrades of the module M to implement a given change becomes unbounded.

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.

This principle is focussing on how data structures are passed to processing functions. Data version transparency is the property that data entities can have multiple versions, without affecting the processing functions that consume or produce them.

Proof: Consider a data structure D, that is passed through the interfaces of N versions of a processing function F. The introduction of a mandatory upgrade of the data structure D will require the adaptation of the code that accesses this data structure for N versions of F, unless D exhibits version transparency. According to the

 ISBN: 978-1-61208-184-7

assumption of unlimited systems evolution, N will increase over time and will become unbounded, and so will the number of versions of the processing function F. As a result, the number of additional adaptations of the code, which interfaces with the data structure, becomes unbounded.

3) Action version transparency:

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

This principle is focussing 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 that call this processing function.

Proof: Consider a processing function P that is called by N other processing functions F. The introduction of an upgrade of P will require the adaptation of the code that calls P in the N functions F, unless the upgrade of function P exhibits version transparency. According to the assumption of unlimited systems evolution, N will increase over time and will become unbounded, and so will the number of calling functions F. As a result, the number of additional adaptations of the code, which are calling the function P, becomes unbounded.

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 focussing 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. The (error) state should be kept in a separate data entity.

Proof: Consider a processing function P that is called by N other processing functions F. The introduction of an upgrade of P, possibly with a new error state, will require the N functions F to handle this error, unless the upgrade of function P exhibits state keeping. According to the assumption of unlimited systems evolution, N will increase over time and will become unbounded, and so will the number of calling functions F. As a result, the number of additional code to handle the new error in each function F, becomes unbounded.

IV. New theorems: entity instances

Modularity is a central concept in systems theory and has played a crucial role in software engineering since the 1960s. Doug McIlroy described a vision of the future of software engineering in which software would be assembled instead of programmed [16]. Studying evolvability of software in terms of its modular structure is widely accepted [1] and modularity is generally associated with use and reuse. Hence, when a module is used more than once during run-time we can call each use an *instance*.

Regev et al. proposed a definition of "Business Process Flexibility" [17]. We derive from this definition a more general interpretation of flexibility-to-change: "the capability to implement changes in a module's type and instances by changing only those parts that need to be changed and keeping other parts unchanged". In this interpretation, we specifically mean that a type change has influence on all upcoming instance creations. Meanwhile the existing (older) instances are not aware of this change, and should not be affected by this change. In other words, we consider the following sequence. An original version of a module's type is compiled on moment t=1. An instantiation of this module is created during the launch of the system on moment t=2. On moment t=3 we compile a new version of the module's type. We start a new part of the system, which is realizing a connection with an existing part. More specifically, on moment t=4 we create a new module instance based on the new version, without affecting the existing original module's instance in run-time (which existed since moment t=2). The instance, which showed up in run-time on moment t=4, should not affect the older instance, which was already launched in run-time on moment t=2.

For some instances of software primitives there are specific reasons to evolve, for other instances there are no such reasons. Moreover, even other instances have reasons to evolve on another way because of other specific (application dependent) reasons. Finally, we end up with an additional non-functional requirement, which can be designed on a similar way as evolvability: *support of diversity*. However, when initially identical instances of primitives evolve to a diversity of instances, the four theorems of Normalized Systems are not enough to prescribe how to manage instances. In search for a solution, we introduce two additional theorems, which focus on entity instances and how different versions can co-exist and used. Note that these new theorems are an *extension* of the theory. However, the prediction of the orignal authors of the existing Normalized Systems theory is not violated: these additions do not fundamentally alter the first 4 principles, they only suggest additional principles [7]. Moreover, the new theorems are *run-time equivalents* of the original theorems 2 and 3 (version transparency theorems). Besides, these existing theorems are proven by a simple *reductio ad adsurdum*, and the new ones are proven by evaluating a possible violation of the original theorems. Future research should provide experience reports in order to find possible empirical confirmations.

We define two additional anticipated changes:

- An additional data entity instance (known entity type version)
- An additional action entity instance (known entity type version)

 ISBN: 978-1-61208-184-7

We further posit the *assumption of unlimited systems evolution in both compile-time and run-time*, namely that the system evolves for an infinite amount of time. Note that the run-time evolution of the system is more complex than the compile-time evolution, because the run-time evolution also includes old instances, from which no module's type definitions exist any more in upcoming compile-time. One can imagine situations where new versions are just extending older version, and consequently the existence of the older module's type stops. Equivalently, the amount of both data entity instances and action entity instances will become (theoretically) unbounded. The amount of versions will become unbounded as well.

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.

Proof (reductio ad adsurdum): When a data entity instance, constructed or generated with an old version is showing up in a more recent action entity instance, the action entity instance has to be able to decide whether it can
a) just process the data instance
b) use default values for non-existing fields
c) not process the request, but setting a 'data type obsolete' status.
Consider this action entity instance, which can not identify the version of the older data instance. Next, this action entity instance attempts consuming a non-existing field in the older data entity instance, and end up in an unexpected and/or stateless behavior. The latter would result in a violation of the fourth theorem (separation of states), and thus also of the postulate of Normalized Systems theory.

This principle is focussing on the interaction between data entity instances and action entity instances. It contributes to the problem that instances of the same data entity can differ in version. When a recent action entity meets an older data entity instance, this action has to know the version of the provided data instance, to be able to treat this instance in a correct way. It should for example not happen that the action entity is performing operations with a recently added data field, which is not available (yet?) in the provided data instance.

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, preferably in a separate data entity instance

Proof (reductio ad adsurdum): Consider an action entity instance, which calls another action entity instance, without being able to identify the version of the called action entity instance. Next, the calling action entity instance requests to perform a non-existing (supporting) task of the called action entity instance, without being able to determine a correct state of the called operation. This would result in a violation of the fourth theorem (separation of states), and thus also of the postulate of Normalized Systems theory.

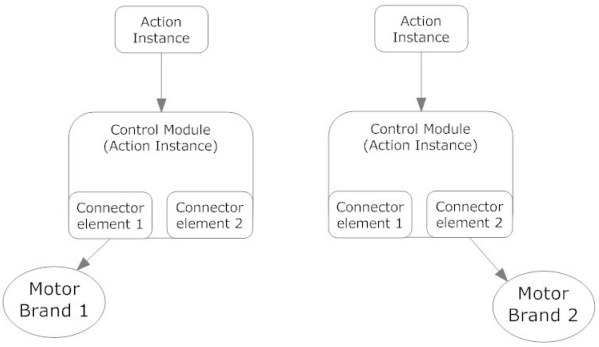

Figure 2. Two instances for different brands

This principle is focussing on the mutual interaction between action entity instances. It contributes to the problem that action instances of different versions are calling other action instances of different versions. The youngest instance has to be able to determine the version of the older instance, to be able to make a good decision concerning the operation of the requested functionality. Consider for example a control module, which is managing the control of a motor [13]. Suppose an existing motor instance is replaced by a motor of another brand, with different system functions. A new version of the control module would support a new connector element, which is managing the connection with the motor of the new brand (Figure 2). To comply to the theorem 'separation of concerns', a new connector element has to be added to the control module. However, besides the property of evolvability, support for diversity is needed too in this case. A calling action has to be able to specify and select which connector should be used in the operation. If, due to the absence of action instance transparency , the wrong (old) connector element is used, we will end up in unexpected behavior of the motor of the new brand.

V. CONCLUSION

Parts of a system, which are initially identical, will differ over time. In other words, some parts will evolve, others not (on the same way). Our ambition is to find rules to facilitate building systems, which are able to support both evolvability *and* diversity.

The Normalized Systems theory contributed in making heuristic knowledge explicit. We think the necessary knowledge for building evolvable systems is available, but often only in the form of tacit knowledge, which is often fragmented design knowledge [10]. The use of this tacit knowledge is very dependent of individuals, or of which individuals are supporting and coaching the development

 ISBN: 978-1-61208-184-7

process. For large systems, it is hardly manageable to only allow developers in the team, who have the required tacit knowledge. Making this tacit knowledge explicit can contribute in facilitating non-experienced engineers to develop evolvable systems.

This paper proposes two additional new theorems, which are in line with the existing theorems. In this contribution we focussed on the existing theorems "data version transparency" and "action version transparency". We specify that these original theorems apply explicitly for written code of data entity and action entities respectively. We suggest to interpret them as "data *type* version transparency" and "action *type* version transparency". Our two new theormes apply explicitly for *instances* during run-time for data entities and action entities. Therefore we call them "data instance transparency" and "action instance transparency".

REFERENCES

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

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

[3] Amin S. M. and Wollenberg B. F. , "Towards a smart grid", IEEE Power and Energy Magazine, vol. 3, no. 5, pp. 34- 41, 2005.

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

[5] Java platform enterprise edition. http://java.sun.com/javaee/.

[6] International Electrotechnical Commission, "IEC 61131-3, Programmable controllers - part 3: Programming languages", 2003.

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

[8] Dijkstra E., "Go to statement considered harmful", Communications of the ACM 11(3), 147-148, 1968.

[9] Mahnke W., Gössling A., and Graube M., "Information Modeling for Middleware in Automation", IEEE Conference on Emerging Technologies and Factory Automation, 2011.

[10] Mannaert H., Verelst J., and Ven K., "Exploring the Concept of Systems Theoretic Stability as a Starting Point for a Unified Theory on Software Engineering", IARIA ICSEA 2008.

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

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

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

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

[15] van Nuffel D., Mannaert H., de Backer C., and Verelst J., "Towards a deterministic business process modelling method based on normalized theory", International journal on advances in software, 3:1/2, pp. 54-69, 2010.

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

[17] Regev G., Soffer P., and Schmidt R., "Taxonomy of Flexibility in Business Processes", Proceedings of the 7th Workshop on Business Process Modelling, Development and Support, pp. 90-93, 2006.

 ISBN: 978-1-61208-184-7