Towards Evolvable State Machines and their Applications

Dirk van der Linden[1], Wim Ploegaerts[2], Georg Neugschwandtner[1], and Herwig Mannaert[1]
[1]University of Antwerp, Belgium
{dirk.vanderlinden, georg.neugschwandtner, herwig.mannaert}@uantwerpen.be
[2]PWCS bvba, Belgium,
wim.ploegaerts@pwcs.eu

***Abstract*—Since several decades, the pressure on organizations to swiftly adapt to their environment has been increasing. At the same time, the complexity of products and services has been growing. One of the consequences is the increasing importance of the evolvability of software, whether this software supports production control systems in industry or business information systems. Over the past decades, finite state machines have become an increasingly popular tool for modelling behavioural aspects of software. 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. This design is discussed in the context of automation systems as well as more general information processing applications.**

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

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. The evolvability of production control and information systems is the primary focus of this paper. We will present a design for evolvable state machines, based on Normalized Systems Theory (NST). Its concepts can be applied to all state machines; examples will be discussed for control systems, kiosk software and business process automation.

Evolvability is a critical non-functional requirement on software. Better evolvability favourably impacts the challenging task of software maintenance, where *adding a six-lane automobile expressway to a railroad bridge is considered maintenance* [2]. In their review of evolvability as a characteristic of software architectures, Ciraci and van den Broek [3] 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.

Maintenance activities often disrupt normal (productive) system operation. If dynamic reconfiguration can be achieved, downtimes of systems can be reduced: a change which can be performed without a complete shutdown is called a 'dynamic reconfiguration' – in contrast to a 'static reconfiguration', which requires the complete shutdown of a system [4]. The ability of an evolving system to introduce a change by dynamic reconfiguration is a special property of the system and the type of change.

Having to stop production for maintenance can be especially costly for continuous production systems or 24/7 production operations. Consequently, production loss or delay of production due to an update of the system must be considered an indirect maintenance cost. In fact, there is a similar cost in information systems, but this cost is often neglected because it is less visible – for example, because it only indirectly affects customer satisfaction. System restarts and maintenance windows have become a generally accepted practice, even for business critical applications. For example, Microsoft Servers require restarting after certain updates of the operating system. But even payment systems are shut down several times per year 'for maintenance', which typically takes place between 00:15 and 02:15 at night in Belgium. On the site of Atos Worldline this is called a 'system stop', and *during these interventions, the payment network is not available and it is not possible to carry out electronic payments* [5]. This can be very inconvenient for the user if the restaurant bill has to be payed while the cards do not work, and neither does the ATM (automatic teller machine). Customer service could be improved if dynamic reconfiguration was made a design requirement for the payment system.

A. Maintainability improvements

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 [6]. The technology shift from hardware to software provided more flexibility and an improvement of maintainability.

Around the same time, dynamic reconfiguration was introduced as a feature of the *Multics* operating system. Multics is an acronym for *Mult*iplexed *I*nformation and *C*omputing *S*ervice. It was a mainframe operating system for which the design and planning started in 1964 [7]. It was commercialized by Honeywell and used till 2000. The goal was to ensure continuous availability of the mainframe running the Multics computing service, even when maintenance of physical components was required [8]. It was possible to switch between different hardware configurations, allowing for the replacement of CPUs (central processing units) and memory modules without switching off the system – resulting in dynamic reconfiguration.

While these certainly were improvements, the characteristic of true evolvability was (and is still) not totally reached. For example, many serious maintenance operations in the Multics operation system did still require a complete restart of the machine [9], and while changing or debugging a few lines of software code is easier than rewiring relay circuits, 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 [10].

B. Standardized programming languages

For the reusability and portability of features from one (sub)system to another, a common programming language which is shared between these systems is an improvement over proprietary approaches. If possible, the use of standardized programming languages is most appropriate. The IEC 61131-3 standard [11] introduced standardized programming languages for PLCs. However, development environments which include compilers for these programming languages are typically proprietary systems and contain vendor-dependent differences.

In the world of information systems development, the impact of standardization is limited. There are thousands of programming languages. According to [12] more than 115 languages were implemented by 1968, while there were already over 1000 that had been used somewhere by 1999. Only a limited number of these have become relevant, but none can be considered as a global de-facto standard. This is confirmed by the different indices that measure programming language popularity, such as Tiobe [13]. According to Tiobe, the most used languages in August 2013 are Java and C, each having a market share of close to 16%, followed by C++ with a 9.4% market share. Several of these programming languages are defined in an associated (international) standard. As an example, there is ISO/IEC 14882:2011 on the programming language C++ [14]. For the Java language, no international standard is available, even if it is currently the most popular programming language. It also should be noted that only few compilers implement an entire language standard and nothing but this standard. An overview of the support for the C++ standard in different popular compilers is given in [15]. A similar situation exists in the world of relational databases. While an SQL standard is defined in ISO/IEC 9075-1:2008, multiple incompatible SQL dialects exist.

These implementation differences are a first obstacle to evolvability. In most cases it is not possible to simply replace a compiler with a new version from another vendor. Unless special middleware or libraries are used for decoupling (for example, Hibernate), database management systems cannot easily be replaced. Sometimes even the upgrade from an old version to the new version from the same vendor may cause run time errors. Similar issues arise when, for example, a web application framework like Apache Struts needs to be replaced by another framework.

C. Standardized approaches to organizing programs and data

The way how data and functionality are organized within the constructs provided by the chosen development environment is a design decision which is often left to the individual developer. However, in order to be able to share these constructs with other developers, common models can be useful: Not only programming languages, but also the way how data and programs are structured can be standardized.

At the design level, a number of standards are available. Their acceptance amongst practitioners is rather low, however. One of the few standards that stand out in terms of use is the Object Management Group's Unified Modeling Language (UML, ISO/IEC 19505). UML is a graphical modeling language for software engineering. The UML specification contains 14 types of diagrams, seven of which are (static) structure diagrams and seven are used to describe dynamic behavior.

One of the common modelling techniques for functionality is the finite state machine (FSM). FSM follow a straightforward syntax of states and transitions. Their formal semantics is well defined and based on a simple though rigorous mathematical model [16]. Generic in nature, FSM can also be combined with application domain knowledge for the purpose of standardisation. For example, the ISA-88 standard [17] 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 evolvability. The research presented in this paper focuses on the design of an evolvable state machine. Such a state machine could be implemented in one or more of the IEC 61131-3 languages, independent of vendor or CPU type. It should be possible to base applications in information systems on this design as well. The design supports dynamic reconfiguration wherever possible.

The remainder of the paper is structured as follows: Section II gives an introduction on finite state machines, the underlying mathematical model and UML state diagrams. Section III explains the basics of Normalized Systems Theory (NST), which offers a formal guideline to system evolvability. This section also introduces an application concept, derived from Normalized Systems theorems, to support the co-existence of different versions of a program element in an evolvable system. Section IV presents three examples of state machines going through subsequent evolution steps. Section V develops an inventory of anticipated changes to state machines, illustrated by these examples. Section VI proposes a set of design rules for evolvable state machines in the context of the concept presented. Section VII discusses implementation considerations, giving special attention to hierarchical state machines, and Section VIII concludes the paper.

II. State machines

The concept of finite state machines (FSM) has its origins in work by McCulloch and Pitts in 1943, as part of their research on human cognition [18]. It was mainly through the work of George H. Mealy and Edward F. Moore, published respectively in 1955 and 1956, that FSM became popular as a design tool for digital systems.

An FSM is an abstract machine that can be used to model and/or specify sequential logic. A FSM 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 its 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.

This definition can be cleanly expressed by means of a simple mathematical model. Following [16], a deterministic finite automaton is represented by the 5-tuple

$$(Q, \Sigma, \delta, q_0, F)$$

with

- $Q = \{q_0, q_1, ...qn\}$, a finite non-empty set denoting the states the system can be in;
- Σ, the input *alphabet,* a set of symbols denoting the possible inputs;
- $\delta \in Q \times \Sigma \rightarrow Q$, the state transition function;
- q_0, the initial state;
- F, a set of final states.

The input alphabet Σ is the finite set of symbols that are accepted and can cause a transition. An input symbol represents an external condition. The state transition function δ determines the next state, based on the current state and input symbol. This is a partial function: The function is not defined for all possible combinations of states and input symbols. It is assumed that the next state will always be reached.

As far as system output is concerned, this model is quite limited: It can only describe an "accept/reject" result. If, for a particular sequence of input symbols, the automaton reaches a final state, the sequence is accepted; otherwise, it is rejected. This is not very useful for modelling the output of PLCs or most other software systems. For these purposes, it is helpful to extend the model so that it can describe the output that the system generates as it transitions between the states. The term 'finite state machine' is typically used to refer to an automaton with such output capability.

Traditionally, two different designs of state machines are distinguished: Moore machines and Mealy machines. In a Moore machine, the output is a function of the current state only, while in a Mealy machine, the output is a function of both the current state and current input. Moore and Mealy machines are special cases of the general model shown above. A Moore machine is represented by a 7-tuple

$$(Q, \Sigma, \delta, q_0, F, \Gamma, \omega)$$

with, just as before,

- Q, a finite non-empty set denoting the states;
- Σ, the set of possible inputs;
- $\delta \in Q \times \Sigma \rightarrow Q$, the state transition function;
- q_0, the initial state;
- F, a set of final states;

and, in addition,

- Γ, a finite non-empty set of output symbols;
- $\omega \in Q \rightarrow \Gamma$, the output, depending on the current state.

The representation of a Mealy machine is the same 7-tuple

$$(Q, \Sigma, \delta, q_0, F, \Gamma, \omega)$$

as for the Moore machine, but with

- $\omega \in Q \times \Sigma \rightarrow \Gamma$, as the output depends on both the current state and the input.

In practice, FSM are defined either in diagram notation (typically circles and arrows), or by a state transition table. In Figure 1, a simple Moore and a simple Mealy machine are shown as a UML (Unified Modelling Language) state

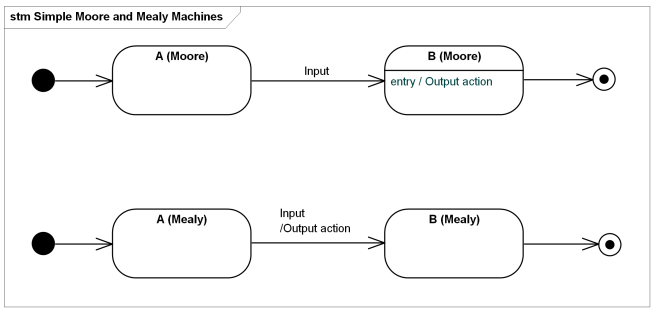

Figure 1. Moore and Mealy machine example

diagram. In UML, states are denoted by round-cornered rectangles, contrary to the classical graphical notations where circles are used. The black circle is the *initial state*, where 'the machine starts'. The circle with the dot is the *final state*. Transitions are denoted by arrows between two states.

For the Moore machine, outputs occur when the machine enters the new state. The transitions fire when the proper input is present. Therefore, the output action is shown on the state itself, and the transition is labelled with the input only. For a Mealy machine, every transition generates an output. Hence, every arc is labelled with both the input and the resulting output.

In a Moore machine, if different transitions take the finite-state machine to the same state, the same action is performed for all these transitions (since output is associated with the state). In a Mealy machine, if multiple transitions that have the same destination state should cause the same action, this action must be triggered by every transition on its own. Still, every Mealy machine can be converted to a Moore machine and vice versa, i.e., their mathematical models are functionally equivalent.

A. Implementation

Finite state machines grew popular in the world of digital circuit design. Both the next state and the outputs are calculated by means of a collection of logical gates (combinatorial logic). Figure 2 shows the respective designs for a Moore and a Mealy machine.

In a Moore machine implementation, the output must be calculated when entering the state. In a Mealy machine, the states are just 'resting points'. No more calculations are needed when the new state is entered. All actions occur during the state transition. In the mathematical models, it is not relevant when the output is calculated, since it is assumed that this calculation is instantaneous. However, the resulting hardware design will be different for a Moore machine and its Mealy equivalent. Typically Mealy machines have fewer states and are faster than Moore machines. Moore machines on the other hand are easier to program and require less output logic. Moore machines are often used for the design of controllers in digital circuits, while most software implementations use the Mealy model or a combined/extended model.

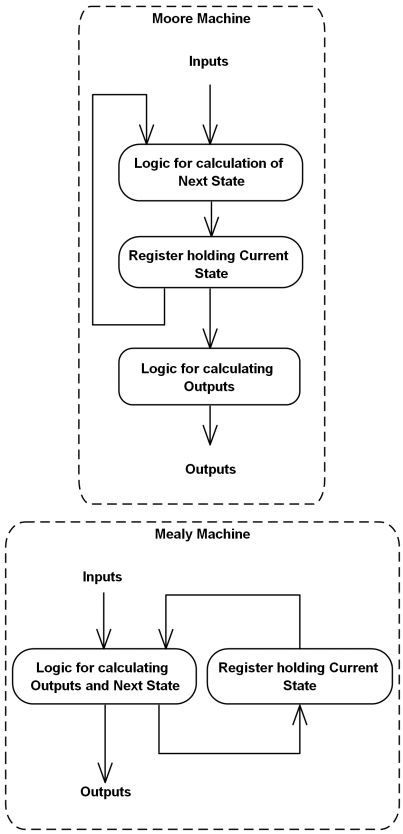

Figure 2. Moore and Mealy machines

State machine implementations must consider how the 'real world' compares to the idealized model. For digital circuits, this is relatively straightforward: It is only required that a stable electrical output is obtained within a single clock cycle. If this is the case, in normal operating conditions, the implementation matches its underlying mathematical model. The properties derived from the model also apply to the digital circuit. Drawing the same conclusion for state machines in a software system in general will likely be more difficult. For example, the model assumes that no error occurs when calculating the next state in response to a trigger. To make the software system reflect the model accurately, either the state machine has to be made more complex to reflect these exceptional conditions, or a number of restrictions need to be imposed on the software system (for example, that calculations should not fail; or, in a Moore machine implementation, a new output should not be produced if the next state is not reached). Either way, a consistent software implementation of the simple mathematical models may turn out to be quite complex.

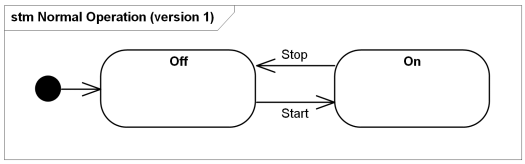

Figure 3. Motor control state machine, version 1

B. State space size and complexity

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. Figure 3 shows a very simple model of motor control system, which could be implemented on a PLC. The motor has two states, 'On' and 'Off' and an initial state. Two conditions affect the state of the motor: the 'Start' and the 'Stop' command. The output (whether the motor runs or not) is fully determined by the state, hence this is a Moore machine.

Typically, a factory contains more than one motor. In case this state machine is extended to control multiple motors with a single controller, its 'state space' increases. The 'state space' is defined as the collection of all possible states the controller or system can be in. The state space of Figure 3 contains two states. In case this state machine is extended to control two motors, there are four states (On—On, On—Off, Off—On, Off—Off); a third motor would yield a state space of eight states. An installation with 100 items to be controlled, each for example having 10 states, yields 10x10x...x10=10^{100} different states, an installation with 1000 items 10^{1000} states. The fact that the number of states grows exponentially with the size of the installation is called *state space explosion*. It clearly is not practical to model a large system this way.

State space explosion is often used as an argument against the use of state machines by software designers who are not familiar with the concept. However, nobody would ever design a 'flat' controller with 10^{1000} states. The solution to this problem is twofold. First, the overall design can be split into a collection of communicating state machines; in our example, a separate instance of the original state machine for each motor. Second, states can be grouped hierarchically.

However, careful design of the communication between the state machines is required: Wagner and Wolstenholme [19] point out that *sending unconstrained messages to another state machine is like a jump to another program part* using a 'goto'. As state machines run concurrently, issues such as deadlocks may arise. In order to create maintainable state machines, the messaging between the state machines needs to be restricted.

C. UML statecharts

In 1987, Harel [20] introduced 'statecharts' as an extension to classical state machines, supporting hierarchy and concurrency in a single graphical formalism. Harel's statecharts form the basis for state machine diagrams in the UML standard, which includes them as a means to specify the behaviour of a software based system. Statecharts introduced hierarchical or *composite* states to enhance the readability of complex state diagrams. A system is regarded as an abstract state machine with a limited number of states. Every state can be further refined by defining another state machine within that state, supporting a top-down design approach. The UML standard does not specify up to which level the states must be refined. Thus, in a UML statechart, 'not all states are equal'. Some states can be modelled down to the level of boolean logic, other states may be a black box implemented by an entire business application.

In UML statecharts, transitions are caused by 'triggers'. The trigger corresponds to the input symbol in the FSM model, but can be any signal, event or change in a condition. The transition can be conditional to a 'guard'. The result of the transition can be twofold. First, the transition can have an 'effect', an action that is executed conditional to the firing of the transition. Second, the transition results in a state transition which can also cause a number of actions to be performed. An *exit* action can occur when the current state is left; an *entry* action may be executed when entering the new state. In UML, the order in which these actions are performed is well defined (exit action first, then transition effect, then entry action). Likewise, when hierarchical states are used, the order in which all actions need to be performed is well defined. A basic assumption is that a state machine can only start processing an event if it has finished processing the previous event.

Apart from the hierarchical definition of states, UML statecharts have a much richer syntax than classical state machines have, such as constructs to model concurrency through forks, joins and regions. Through the use of 'pseudo states' such as a junction, an entry and exit state and even a 'history state', the communication between the different state machines in a design can be specified.

Because of this added complexity, a formal definition of the semantics of UML state diagrams is much less straightforward than the simple mathematical model defining the semantics of finite state machines. This almost necessarily results in a number of ambiguities in the specification. Regarding the latter, [21] reports that 29 new unclarities were introduced in the UML 2.0 state machine specifications. In [22], an axiomatic semantics of these UML 2.0 state machines is provided by giving solutions outside the UML standard. According to [23], the current UML standard 2.4 introduced 6 aditional ambiguities.

In this paper, we study design rules for evolvable state machines. Even though we make use of UML, it is not our intention to analyse and discuss all the syntactical features available in UML statecharts. Our use of UML is limited to the capabilities of a classic FSM extended with simple

composite states. UML statechart constructs dealing with concurrency and complex communication will be the subject of future research.

III. Normalized Systems Theory

Software undergoes an ageing process, as recognized by Parnas [24]. Since there are indications that this ageing process is also happening with business processes [25], 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.

Earlier work pertaining this subject matter was done by M. Lehman, resulting in his 'Laws of software evolution'. He formulated the law of increasing complexity, expressing the degradation of a system's structure over time [26]:

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

Based on his work for IBM on the design and implementation of the OS of IBM mainframes in the late 1960s, Frederic Brooks [27] made the observation that

> *"Program maintenance is an entropy-increasing process, and even its most skilful execution only delays the subsidence of the system into unfixable obsolescence."*

Ever since, a myriad of software engineering methodologies were invented, new programming languages were created, paradigms developed from 'structured' over 'object oriented' to 'aspect oriented' programming. This did not, however, fundamentally change the issues related to software evolvability. Brooks stated that because of the very nature of software *no inventions will do for software productivity, reliability, and simplicity what electronics, transistors and large-scale integration did for computer hardware* [28]. Part of the complexity is caused by constant pressure for change, imposed by continuous change of the environment in which the software system is embedded.

In software development, every change causes a further deterioration that 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 [29]).

The theory of Normalized Systems was introduced to challenge Lehman's law [30]. 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" (these are are widely accepted heuristics, e.g., [31]), 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,

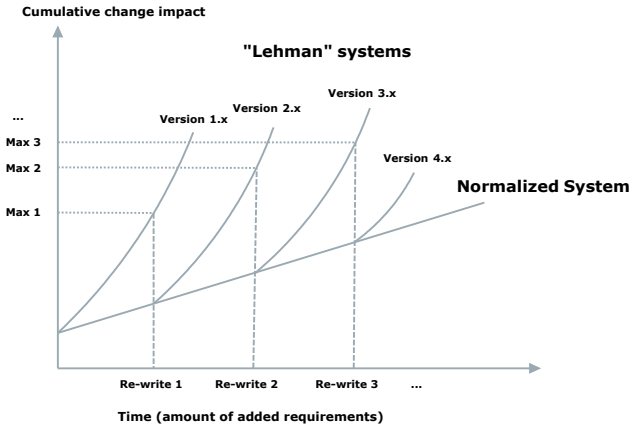

Figure 4. Improving software structure with a re-write [29]

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 [32]. 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.

A. Applying instance version diversity

In order to create an evolvable software system, the above-mentioned rules must be respected throughout the entire design and implementation. In the following, we introduce an architecture supporting version diversity in PLC systems as an application case study. It is designed to support the implementation of cross-vendor PLC systems, allowing for the co-existence of multiple versions of all hardware and software components, and to support true dynamic reconfiguration. Version diversity is supported with respect to:

- Application functionality
- Vendor dependencies in PLC programming
- Vendor dependencies regarding the control of field devices (e.g., a motor).

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 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.

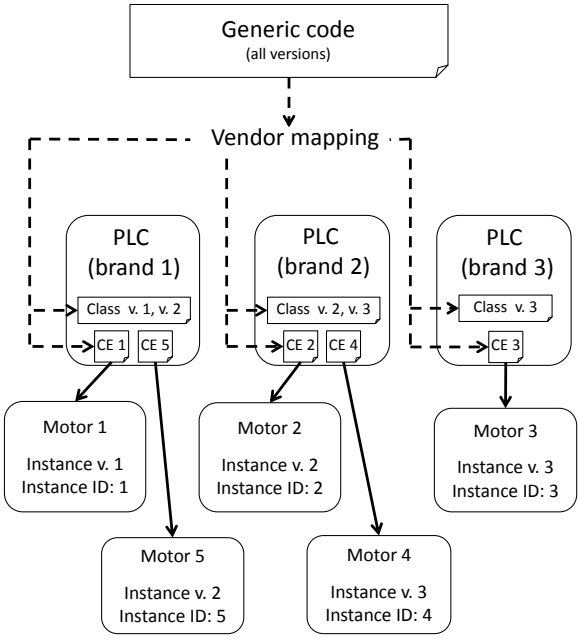

Figure 5. A generic software module with heterogeneous instances

The architecture presented intends 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. 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 [11]). 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

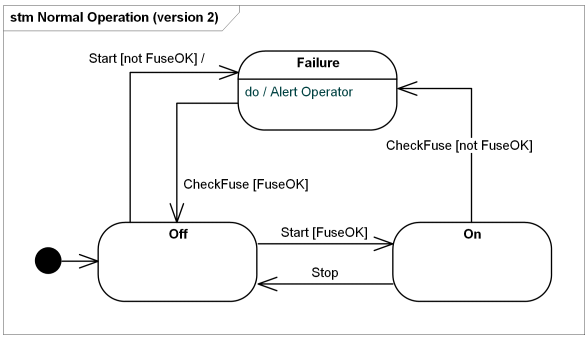

Figure 6. Motor control state machine, version 2

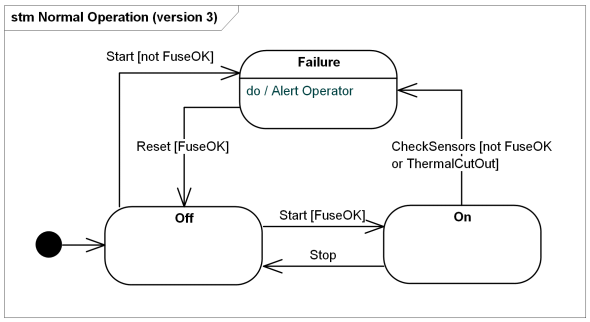

Figure 7. Motor control state machine, version 3

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.

IV. State machines and evolution

In this section, we discuss examples for how state machines could evolve in response to new requirements and hardware capabilities, leading to new versions of the state machine.

A. Motor control

The state machine in Figure 3 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.

In version 2, the start condition becomes 'Start and FuseOK' (Figure 6). In UML syntax, we model this by extending the 'Start' trigger with the 'FuseOK' guard, making the transition conditional on the state of the fuse. The additional state 'Failure' is introduced. The condition for transitioning from the 'Off' state to the 'Failure' state is 'Start and not FuseOK', i.e., the transition is triggered by the 'Start' event, also conditional to the state of the fuse. If the fuse blows during the 'On' state, a transition to the 'Failure' state results. To detect the state of the fuse, a 'CheckFuse' function is needed, being the trigger for the transition from the 'On' state to the 'Failure' state. In addition, when entering the failure state, a notification is made to trigger an operator to solve the issue, represented by the action 'Alert Operator'. The condition to go from the 'Failure' state to the 'Off' state is 'FuseOK', the trigger is 'CheckFuse'. This implies that the engine will restart automatically when the fuse is repaired.

The third version considers the situation that the motor can stop due to a thermal cut out in the 'On' state (Figure 7). This is modelled 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. Hence, an additional 'Reset' trigger is introduced resulting in a new version of this transition.

In version 2, only the state of the fuse needed to be checked, by means of a function 'CheckFuse' triggering the transitions from the 'On' state to the 'Failure' state and from the 'Failure' state to the 'Off' state. In the third version, two conditions need to be checked: the state of the fuse and a heat sensor connected to the motor. Depending on the implementation, this may require the creation of a new version of the trigger that can deal with multiple sensors.

B. ATM example

This example introduces the use of hierarchical states in UML state machine diagrams. It shows a high level specification of an ATM from a user perspective, limited to the withdrawal of money (Figure 8). As in the motor control example, two alternative versions of the initial design will be discussed, each resulting from changing requirements.

The process modelled in this example starts when a user inserts a bank card. This user must first authenticate. If the correct PIN (personal identification number) code is supplied, money is dispensed (Figure 8). The state machine

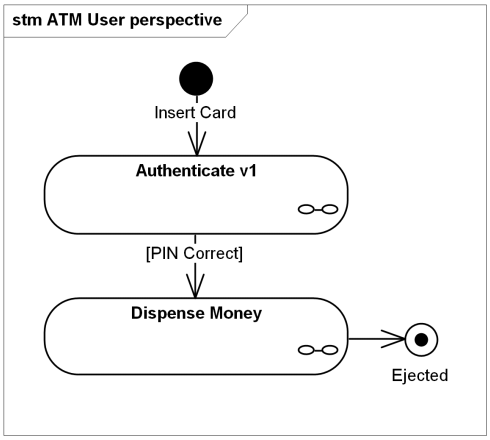

Figure 8. ATM specification using composite states

has two states: 'Authenticate' and 'Dispense Money'. The transition between the states is triggered by 'CheckPIN', a function which is started for example after having pressed the 'OK' button on the ATM. In this very simple example, the user is not given the option to select the amount of money withdrawn.

There is a major difference with the previous state machine examples. Both states in these examples are not simple, fully defined states as in the motor control example. They are in fact state machines themselves. Both states are therefore modelled as composite states, indicated by the 'infinity symbol' in the example. In Figure 9, detail for the 'Authenticate' state is provided.

The authentication starts with the processing of the card. When the machine is in the 'Read Card' state, the card is processed by reading and decoding the chip or magnetic strip on the card. Once the card is processed, the Authenticate state machine will automatically transition to the 'Enter PIN' state. Note that no trigger or guard is specified on this transaction.

When the machine arrives in the 'Enter PIN' state, the entry action of the state is executed. As a result, the machine will ask the user to enter a PIN code. When the user presses an 'OK' button on the ATM, 'PIN Entered' is triggered and the PIN code will be validated using the 'check PIN' action. If the PIN code is not correct, the user must try again, else the final state of the 'Authenticate' super state will be reached. When this final state is reached, the 'Authenticate' state itself is considered as complete and the system can transition to the 'Dispense Money' state. Once the money is dispensed, the card will be ejected and the final state of the overall process is reached.

The diamond within the 'Authenticate' state machine is a 'choice pseudostate'. This state machine could also be drawn with two seperate transitions with the same trigger as in the motor control examples above.

In this version, the ATM has serious drawbacks. First,

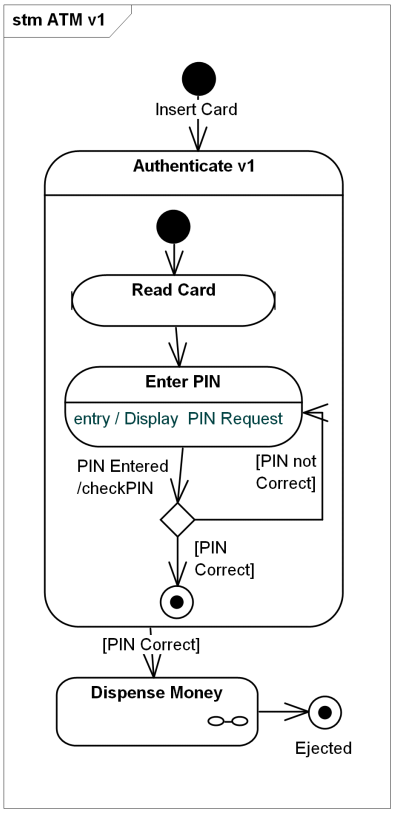

Figure 9. ATM version 1

the user can supply an unlimited number of PIN codes. Second, the card will only be returned after having supplied the correct code. To amend the first problem, the concept of an 'error counter' is added. Two actions to increase and reset this counter are added. If the card is unreadable or if the maximum number or retries is reached, the card is ejected. The user is also given the opportunity to cancel the session and eject the card (Figure 10). In these three cases, the 'Authenticate' state machine transitions to the state 'Transaction cancelled', modelled as an exit state. At the higher level in the state machine hierarchy, this exit state unconditionally transitions to the final state 'Ejected'.

In both versions the internals of the states 'Read Card' and 'Enter PIN' are not specified. To implement the ATM system, more detailed specifications are required, possibly resulting in additional levels of state machines and sub-states. Changing requirements may cause an evolution of the internals of these states, resulting in a third version of the ATM example.

As mentioned above, in version 2, 'PIN Entered' was triggered by pressing an 'OK' button on the ATM. Assume that in a third version of the ATM, the use of this button should be eliminated if the number of digits of the PIN codes can be derived from the card type, e.g., 4 digits for a Belgian bank card. The 'Read Card' implementation should

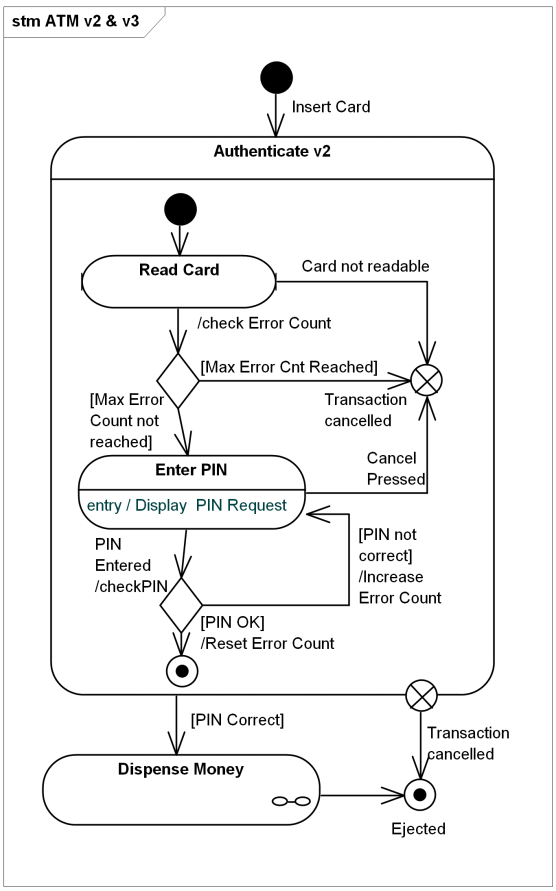

Figure 10. ATM versions 2 and 3

be enhanced to determine the type of card. The 'Enter PIN' implementation should be changed such that

- the entry action becomes conditional on the card type ('Enter PIN' or 'Enter PIN and press OK');
- for a Belgian card the 'PIN Entered' trigger fires once the fourth character is entered, and
- for any other card, pressing the 'OK' button causes this trigger to fire.

In this new version, the 'Authenticate' state machine itself does not evolve to a new version, only the two sub-states 'Read Card' and 'Enter PIN' are changed. Also, a new parameter needs to be passed, since the card type is detected by 'Read Card' and must be used by 'Enter PIN'. However, these differences do not lead to a change in the diagram. Therefore, Figure 10 does not change for version 3 of the ATM.

As in the motor control example, it is possible that both versions of the sub states need to co-exist for a certain time. Assume that the 'Enter PIN' process requires a software update, while the adaptation to 'Read Card' requires the physical installation of new hardware. Ideally, the software is upgraded on all ATMs, irrespective of the version of the

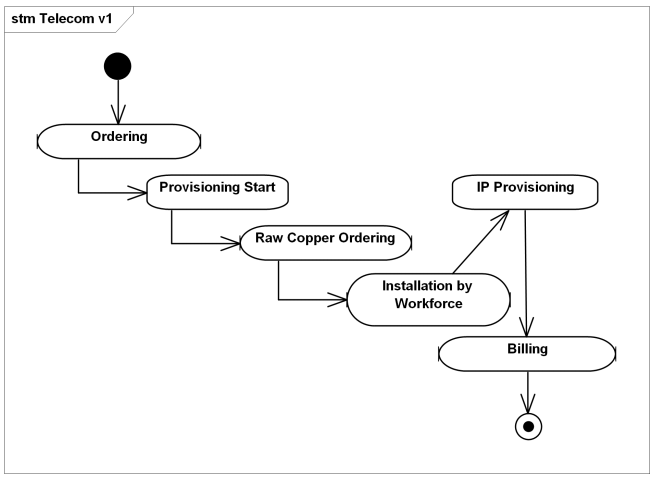

Figure 11. Telecom process, version 1

card reader.

C. Integration of COTS applications

In a third example, additional requirements on the evolvability of state machines will be illustrated. This example is based on the experience of one of the authors with a greenfield software implementation and integration for a Dutch telecom operator. Some of the outcomes of this project for the academic community are documented in [33] and [34].

Running a telecom operation requires the automation of numerous very complex business processes. Most operators resort to the installation of Commercial Off-The-Shelf (COTS) applications instead of implementing everything in-house 'from scratch'. Often multiple COTS applications are installed, such as a Customer Relation Management (CRM) system, a telecom billing system, fraud detection, a network inventory system, an ERP system, etc. These COTS applications are large software systems, the installation of which requires extensive parametrization and configuration to implement the desired business processes. Typically, a multi-million dollar budget is required for the installation and implementation of a single COTS application.

To create end-to-end workflows supporting the different business processes, the different COTS applications must be integrated, this being a bespoke development for the operator. The integrated workflows can be modelled using hierarchical state machines. The highest levels of the hierarchy are part of the bespoke development by the operator. At a certain level in the hierarchy, a substate will be fully contained in one of the COTS applications.

Figure 11 shows a simplified order-to-bill process for a new DSL line, stripped from all triggers, guards and actions. In this example, an order for a new DSL line is treated first in the CRM system. Once the order is completed, it needs to become available in a 'service provisioning system'. The

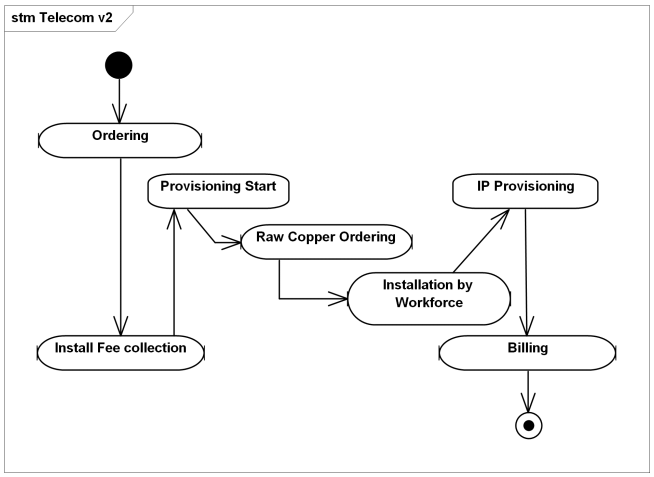

Figure 12. Telecom process, version 2

latter is used to manage the installation of the new line and ensure that the appropriate services are provisioned. The provisioning system needs to communicate with a system of another operator to order raw copper. After delivery of this line, the DSL line is physically installed at the customer premises, a process that is managed by means of a 'workforce management system'. Next, the IP service is provisioned using the service provisioning system. Finally, once, the system must ensure that the appropriate billing information is created in the telecom billing system.

In Figure 12, a second version of this process is shown. In this adapted process, the customer must pay the installation fee before the provisioning starts. Once the service is provisioned, the recurring part of the billing must be activated. When the state machine arrives in the 'Install Fee collection' state, it will hand over the control to the billing system where an invoice will be created and sent to the customer. Next, the entire process will halt, waiting for the customer to pay. Only when the appropriate amount is collected, control will be handed over again to the higher level state machine.

The evolution of version 1 to version 2 is a complex process:

- The entire software installation runs on a large number of servers; the implementation of updates to any of the COTS systems is complex and time consuming. It is nearly impossible to shut down and restart everything at the same time without affecting user experience and/or operations. Attaining the ability of 'dynamic reconfiguration' would be a significant advantage.
- Every substate in the process requires a considerable period of time to complete. During the execution of the overall process for any given order, one or more updates and new versions may be introduced. Depending on a number of criteria, existing orders may still need to be handled according to the the old version. For example, the installation fee can only be collected before the installation if this was required during the ordering phase; if not, the order will need to be processed using version 1 of the state machine.

V. Changes to state machines

Industry usually prefers the Mealy state machine model over the Moore model, possibly because a Mealy machine often ends up with a lower amount of states, and is thus more compact. In contrast, Normalized Systems Theory advocates a higher granularity of modules. If 'more compact' results in a lower amount of (larger) modules in case a Mealy machine is implemented, this would favour the Moore type.

Also, a Mealy machine combines transitions with actions, which can result in violations of the separation of concerns theorem: When a new version of a transition is implemented in the same module as the corresponding, resulting action, both original and new version need to co-exist. When a new version of the action is introduced afterwards, this update has an impact on both versions of the co-existing modules containing both actions and transitions. Moore machines explicitly separate states and transitions. For the implementer, it is easier to comply with the separation of concerns theorem if states and transitions are separated in the model the software is based on.

The preference of industry for Mealy machines is understandable: a lower amount of modules probably results in a lower implementation effort due to the lower amount of module definitions and module interfaces. However, changes to a single transition or to a single action might result in combinatorial effects when transitions and actions are implemented in the same module.

In order to expand on these general observations regarding the evolvability of state machines, this section is dedicated to a detailed examination of anticipated changes, illustrated by the examples in the previous section. As already indicated at the end of Section II, only a subset of the constructs available in a UML statechart is studied.

A. Identifying singular change drivers

The following changes can be observed for 'flat', single level state machines:

- adding a state (e.g., 'Failure' in Figure 6)
- adding a transition (e.g., 'Card not readable' in Figure 9)
- changing a state (e.g., change of 'Enter PIN' entry action from ATM version 2 to 3)
- changing a transition (e.g., the 'On' $\rightarrow$ 'Failure' transitions in Figures 6 and 7).

Neither adding/changing a state nor adding/changing a transition are singular actions, since they may involve a number of independent actions. Adding a transition may require the addition of a trigger, a guard and an action. Adding a state may require the addition of two transitions, an entry

and an exit action. Changing a state may require additional data to be passed between states. According to NST, change drivers need to be singled out to study evolvability. Hence, the above-mentioned anticipated changes need to be further refined.

Likewise, neither changing the direction of a transition nor changing the target of a transition are independent actions. Both consist of adding a new transition together with simultaneously deleting the original transition. As will be discussed below, deletions are not desirable from an evolvability viewpoint, which is why they are not included in the list above.

For reasons of simplicity we will not further take the 'guard' of a transition into account. It is assumed that the trigger event will occur conditional to the value of the guard. We consider this as a further restriction on the subset of UML state machine constructs that are within the scope of our current analysis. A detailed analysis of this subject matter shall be part of future research.

While triggers and actions can be easily distinguished as singular change drivers, it is more difficult to define a 'singular addition of a transition'. In its most basic form, a transition $A \rightarrow B$ unconditionally fires when the state A becomes current, and causes no effect other than putting the state machine in this next state B. On arrival in state B, actions related to this state will of course be executed.

The addition of an unconditional transition to an existing state machine may cause inconsistencies, however. Assume, for example, that the transition from 'Read Card' to 'Transaction cancelled' in Figure 10 were unconditional. The resulting machine would always transition to the final state 'Ejected', which is of course, undesirable. There are many solutions to this problem. A possible solution, requiring the introduction of concurrency in the design, is the following: Immediately after reading the card, execution should split in two parallel, concurrent paths. A first path goes to an end state by ejecting the card. In parallel, the process continues as if the card was read. While this change to the design of the state machine yields the desired result, it clearly is not a singular change.

To find a definition for the singular addition of a transition, we shall resort to the mathematical model underlying the state machines. In this model, the state transition function

$$\delta \in Q \times \Sigma \rightarrow Q$$

maps the combination of a current state and an input to a next state. Not every combination (current state, input) is defined. Only a limited number of inputs are accepted in any given state. Every combination (current state, input) can only occur once, i.e., the next state is defined unambiguously.

In this model, adding a transition cannot introduce an inconsistency in the state machine as long as the restriction is met that the next state is defined unambiguously. We therefore impose additional restrictions on the UML state machines we consider in order to replicate this characteristic, and limit ourselves to state machines without concurrency that only have conditional transitions. Hence, a singular change driver can be obtained when the transition has a trigger and under no circumstances a condition could arise where two transitions need to fire concurrently. Note that the same restriction holds for every addition and change of a trigger.

These are significant constraints on the usual semantics of UML statecharts. Nonetheless, in the remainder it is implicitly assumed that these consistency requirements are respected through every step in the evolution. Hence, they will not be part of the design rules we will propose.

This results in the following anticipated changes to a flat state machine:

1) adding a transition (maintaining the consistency requirements) between two states, without effect
2) adding a trigger to a transition (changing/deleting)
3) adding an action to a transition (changing/deleting)
4) adding an empty state, without internal actions
5) adding an entry action (changing/deleting)
6) adding an exit action (changing/deleting)
7) passing additional data between states.

In case of non-flat, hierarchical state machines, the internal behaviour of the states will become more complex. In the telecom example, the 'Billing' state, comprising a very complex state machine in a COTS application, was split in two parts. Part of the actions executed in the state 'Billing' will be separated amongst the new states. Part of the action, such as the creation of the invoice and the collection of the payment, will need to be supported in both states. A number of anticipated changes can be derived:

1) adding a state machine to a flat state
2) changing an existing state machine
3) deleting a state machine to return back to a flat state.

For the purpose of our discussion, we only consider a restricted set of hierarchical state machines which can be flattened. Therefore, composite states only have a notational purpose. Since changes to them can be expanded to changes on the equivalent flattened state machine, they are not further considered as anticipated changes. If certain restrictions are lifted, and e.g. concurrency is allowed, however, this will need to be reconsidered.

Apart from the entry and exit actions of a state, the UML standard also specifies 'internal transitions'. These are transitions that cause an output action but do not cause a state transition. These transitions fire as a response to a specific event if the machine is in a given state. This type of transitions has deliberately been excluded from the discussions above. We consider them as special cases of hierarchical state machines. From a semantical point of view, this is a further restriction we impose. It will, however, not impact the conclusions listed below.

B. Changes and version transparency

Deleting a state, transition or any other element usually is intuitively considered an anticipated change to a design. However, according to NST, a deletion violates the action version and data transparency theorems and should therefore never be allowed. Assume that in the ATM example, the bank card technology evolves in such a way that the number of digits to be entered can be read from the card using a new card reader. This of course requires an update of the card reader which is used in the 'Read Card' state. In a fourth version of the state machine, 'Enter PIN' would then make use of this number rather than the card type. If, however, the card type parameter is deleted, versions 3 and 4 cannot coexist any more. Similar reasoning holds for the deletion of transitions and states. Changing the direction of a transition is in essence (a) the addition of a new transition in the opposite direction and (b) the deletion of the original transition, and should therefore not be allowed either.

To keep systems compact nonetheless, NST proposes an extended 'garbage collection' approach on the design. The removal of states, transactions, or other constructs can make it impossible for older versions to exist. Consequently, deleting a construct should only be allowed if one can guarantee that no existing module makes use of it. In the ATM example, this means that the card readers have been physically upgraded for all ATM machines in the field. Note that the knowledge whether this has been done or not *cannot* be derived from the design itself.

Just like a 'delete', a 'change' can cause a violation of the version transparency theorems. Note that confusion can arise with restricting 'change', because the area of concern of NST is 'change' of an evolving system without causing combinatorial effects. In this paper the theory is applied to state machines. The goal of NST is to determine how a design and/or implementation can be *changed* in a way that a continuous and potentially infinite evolution of the system can be achieved. To avoid this confusion, we prefer to use the term 'modification' when we refer to an update of the functionality or data structure of a construct. To prevent violations of the version transparency theorems, modifications should be based on the concepts of transparent coding or version wrapping [35]:

1) Transparent coding: Transparent coding is defined as inserting internal code in a module which does not affect the functionality of previous versions. Since Normalized Systems require high granularity, it is not unexpected that the individual (small and straightforward) modules or sub-routines end up to be a simple piece of code, on which the programmer has a clear overview. In such cases, the programmer can preview the effect of a functional change on previous versions, and maintain downward compatibility. If the change is not contradictory with one of the previous versions, it might be possible to apply transparent coding. This means that the new functionality can just remain in the module without affecting the original code, even if a calling entity is not aware of the new functionality.

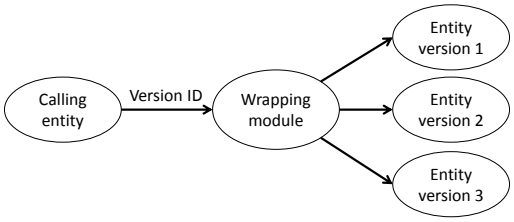

Figure 13. The concept of version wrapping

2) Version wrapping: There will be lots of cases where transparent coding is not possible, because the code is too complex for the programmer to have a reliable overview, or if the new functionality is contradictory with one or more of the previous versions. To exhibit action version transparency, the different versions can be wrapped. The calling action has to inform the called action which version should be used by way of a version tag. In addition, following the separation of states theorem, the called action has to inform the calling action whether the instance of the called action is recent enough to perform the requested action version.

An action entity which is designed according to the concept of version wrapping aggregates functionality for all the versions as separate action entities. Each of the nested action entities contains a version of the core functionality. Following the separation of concerns theorem, the wrapping action entity should not contain any core functionality, but limit itself to wrapping the versions as a kind of supporting task. Following this principle, different versions of a module co-exist in parallel. Figure 13 illustrates how a wrapping module selects the desired version based on the version ID.

C. Anticipated changes

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
- An additional entry/exit action of a state
- A new version of an entry/exit action of state
- A new effect (action) of a transition
- A new version of an effect (action) of a transition.

As mentioned above, adding a transition or adding a state may result in state machines behaving inconsistently. The introduction of a new transition or a new version from a state 'A' may impact all transitions having 'A' as current state, as the conditions on these transitions may need to be changed for consistency reasons. Hence, new versions of all these transitions will need to be created. Similarly, adding a new state 'N' requires the addition of a new transition from a state 'A' to 'N' and from 'N' to some other state, unless if 'N' is a final state. This implies that new versions may have to be created for all transitions that leave from state 'A'. In order to arrive at a new, consistent state machine, multiple of the abovementioned anticipated changes may be required. If however, the implementation of every single anticipated change respects the rules of NST, the overall change will as well.

VI. Derived rules for 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 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.

In UML, a trigger is an action responding to an event that occurs outside of the state machine. According to rule S1, the state machine only contains a generic mechanism to determine what transition to execute based on the contents of a data element.

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

The instance ID allows the connection entities to map this instance to the underlying system. In a PLC, this mapping would contain the hardware addresses; in a business process implementation, the mapping could involve the details of interfacing with a COTS system. The mapping is necessary so that changes in the underlying system (e.g., on hardware inputs) are reflected as changes in the 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 to command the underlying system to perform the action associated with a state of the state machine (e.g., setting the required hardware outputs).

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 that in the motor control state machine example in Figures 3 and 6, 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 (Figure 7) 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.

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

As explained in the previous section, it is a general rule in NST that deletions should not be allowed. Due to its importance, this is reiterated as a separate rule for state machines.

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 version wrapping as discussed in Section V-B.

S6.a *Entry actions of states will be implemented in a separate action entity.*

S6.b *Exit actions of states will be implemented in a separate action entity.*

S6.c *Transition effects will be implemented in a separate action entity.*

Entry actions, exit actions and transition effects determine the output of a state transition. The desired output of a transition can evolve separately from the design of the state machine itself. The three different types of actions resulting from a transition can also evolve independently, even when resulting from the same transition. According to the separation of concerns principle, they should therefore be separated.

VII. Implementation Considerations

The architecture introduced in Section III-A is excellently suited to supporting co-existing, diverse versions of an evolvable state machine. For example, considering the scenario shown in Figure 5, 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.

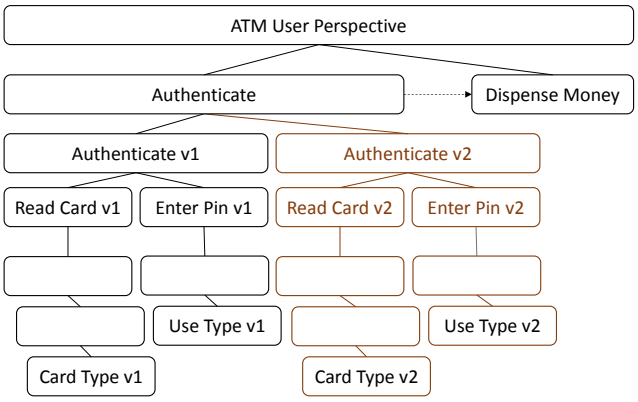

Figure 14. Non-evolvable implementation of the ATM

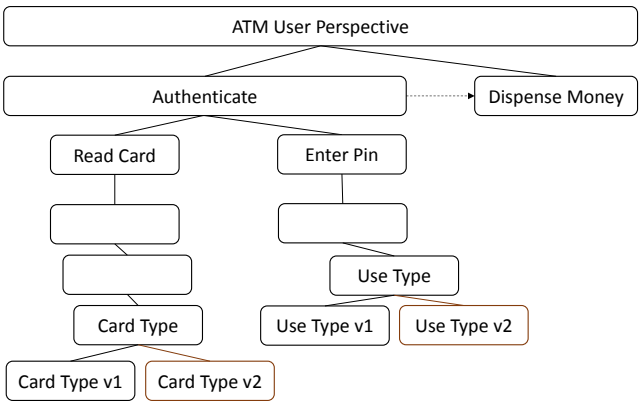

Figure 15. Evolvable implementation of the ATM

A similar approach can be used for the creation of evolvable information systems, which typically have a hierarchical nature. The use of hierarchical state machines makes the situation more complex, especially with respect to the implementation of version transparency. According to NST, changes must remain local. If a single change requires making changes in multiple places, the system is not evolvable.

Figure 14 shows the ATM example, with the different state machines represented in a hierarchy. The top level state machine contains two state machines 'Authenticate' and 'Dispense Money', the first of which contains the state machines 'Read Card' and 'Enter PIN'. To evolve from version 1 to version 2, both state machines need to be adapted. Hence, because of a single change in the requirements, two state machines need to be adapted. Version 1 of 'Enter PIN' can only be used with version 1 of 'Read Card', and version 2 of 'Enter PIN' only with version 2 of 'Read Card'. One could therefore decide to create two versions of the 'Authenticate' machine, each grouping a consistent pair of state machines.

This implementation would not yield the desired results.

The generic implementation of 'Authenticate' cannot decide which version needs to be used. Hence, it cannot be implemented using version wrapping or transparent coding. Moreover, this implementation is not evolvable according to NST. Both 'Read Card' and 'Enter PIN' are hierarchical state machines themselves, each comprising multiple levels of state machines. The change in 'Read Card' would be implemented by changing a state machine somewhere deep in the hierarchy, e.g., the 'Card Type' machine. The card type is consumed by the 'Use Type' machine somewhere in the hierarchy of the 'Enter PIN' machine. A single change would therefore imply the creation of new versions of all the state machines in the hierarchy, creating a non evolvable implementation.

Figure 15 shows an evolvable design where the changes remain local. New versions of the two machines that were changed are created. All other state machines are not impacted by the change. Assume that in the software implemented on an ATM, both version 1 and version 2 remain available. In that case 'Use Type' must be able to determine which version of 'Card Type' was used, as required by the version instance transparency law. This version cannot simply be passed through the hierarchy as this would again require the creation of new versions of all states machines in this hierarchy.

This can be solved by adapting the generic design of a state machine such that every state machine hierarchically enclosed in a state of a given state machine has access to the data element that contains the state and trigger information. The evolution from version 1 to version 2 would thus involve, apart from the evolution of the state machines themselves, a change to the data element containing the state of 'Authenticate' by adding a version number or the card type (or both).

VIII. Conclusion

The state machine is a valuable artefact for modelling systems. 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; similarly, changes in a business environment may require the introduction of new states, new transitions or changes to existing transitions in one or more state machines used in an information system. However, when systems evolve, it follows from Lehman's law of increasing complexity that their further evolution is restrained when the systems' size increases over time.

State machines benefit from the introduction of modularity, as can be seen in the popularity of (hierarchical) UML statecharts. Normalized Systems Theory offers a theoretical foundation to achieve evolvability in modular structures by imposing restrictions on the definition of the modules and their interfaces.

This paper presented a design for evolvable state machines that can be used in both automation systems software and in information systems. The design is based on 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. In an automation system, 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. A similar conclusion holds for information systems in general if the restrictions imposed in this paper are adhered to.

Still, creating software that strictly adheres to the design rules proposed in this paper is a tedious, time consuming and error prone activity. This is, in fact, a major criticism often made in relation to Normalized Systems Theory. Many practitioners claim that it is not possible at all to create such software, except for small applications used for demonstration purposes only, i.e., that Lehman's law applies to all real world software systems.

From the beginning, this problem has been recognized as a major research question related to Normalized Systems Theory. Apart from the development of the theory, it has extensively been investigated how Normalized Systems compliant software can be built. For this purpose, a number of software patterns have been defined [36] [37], together with a set of pattern expanders that allow expanding a higher level description into NST compliant software. After an initial, academic, proof of concept, the development of the expanders became the core activity of the Normalized Systems Institute. The latter is a cooperation of the University of Antwerp with a number of industrial partners and government institutes. By now, the partners in the institute have developed a number of small and mid-sized information systems that are being used in a production environment, proving that it is possible to defeat Lehman's law in a real world software system.

Regarding future work, state machine libraries and toolkits should be improved by adding constraints to follow the rules presented in this paper, resulting in increased system evolvability by ensuring compliance with the theorems on Normalized Systems. The analysis still needs to be expanded to aspects of concurrency and the related UML constructs. Either would the rule set need to be revisited, or it needs to

be demonstrated that real world information systems can be built based on state machines without these constructs.

REFERENCES

[1] D. van der Linden, G. Neugschwandtner, and H. Mannaert, "Towards evolvable state machines for automation systems," in *Proc. 8^{th} Intl. Conf. on Systems (ICONS)*, 2013, pp. 148–153.

[2] P. Stachour and D. Collier-Brown, "You don't know Jack about software maintenance," *Communications of the ACM*, vol. 52, no. 11, pp. 54–58, Nov. 2009.

[3] 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.

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

[5] (Retrieved in 2013) Atos Worldline, System stop. [Online]. Available: http://be.worldline.com/index/en_US/5253162/0000/System-stop.htm

[6] W. Stoecker, "Programmable controllers for industrial refrigerant plants," *International Journal of Refrigeration*, vol. 4, no. 6, pp. 329 – 334, 1981.

[7] E. I. Organick, *The Multics System: An Examination of Its Structure*. Cambridge, MA, USA: MIT Press, 1972.

[8] F. J. Corbató, J. H. Saltzer, and C. T. Clingen, "Multics: the first seven years," in *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, 1972, pp. 571–583.

[9] (Retrieved in 2013) Multics: Myths. [Online]. Available: http://www.multicians.org/myths.html

[10] 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.

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

[12] R. H. Follett and J. E. Sammet, "Programming language standards," in *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., 2003, pp. 1466–1470.

[13] (Retrieved in 2013) TIOBE Software, TIOBE Programming Community Index for September 2013. [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[14] (Retrieved in 2013) ISO/IEC JTC1/SC22/WG21, C++-Standards. [Online]. Available: www.open-std.org/jtc1/sc22/wg21/docs/standards

[15] (Retrieved in 2013) The Apache Software Foundation, Status Of C++ 0x Language Features in Compilers. [Online]. Available: http://wiki.apache.org/stdcxx/C++0xCompilerSupport

[16] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 2006.

[17] ISA, *Batch Control – Part 1: Models and Terminology*. ANSI/ISA-88.01, 1995.

[18] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," in *Neurocomputing: foundations of research*, J. A. Anderson and E. Rosenfeld, Eds. Cambridge, MA, USA: MIT Press, 1988, pp. 15–27.

[19] F. Wagner and P. Wolstenholme, "Misunderstandings about state machines," *IET Computing & Control Engineering Journal*, no. 4, Aug.–Sep. 2004.

[20] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[21] H. Fecher, J. Schönborn, M. Kyas, and W.-P. de Roever, "29 new unclarities in the semantics of UML 2.0 state machines," in *Proc. 7^{th} Intl. Conf. on Formal Methods and Software Engineering*. Springer-Verlag, 2005, pp. 52–65.

[22] K. Lano, *UML 2 Semantics and Applications*. New York, NY, USA: John Wiley & Sons, 2009.

[23] S. Liu, Y. Liu, É. André, C. Choppy, J. Sun, B. Wadhwa, and J. S. Dong, "A formal semantics for the complete syntax of UML state machines with communications," in *Proc. 10^{th} Intl. Conf. on Integrated Formal Methods (iFM'13)*, vol. 7940. Springer, 2013, pp. 331–346.

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

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

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

[27] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 1978.

[28] F. P. Brooks, Jr., "No silver bullet: Essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987.

[29] 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.

[30] 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.

[31] I. Vanderfeesten, H. A. Reijers, and W. M. van der Aalst, "Evaluating workflow process designs using cohesion and coupling metrics," *Computers in Industry*, vol. 59, no. 5, pp. 420–437, 2008.

[32] 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.

[33] M. Snoeck and C. Michiels, "Domain modelling and the co-design of business rules in the telecommunication business area," *Information Systems Frontiers*, vol. 4, no. 3, pp. 331–342, 2002.

[34] W. Lemahieu, M. Snoeck, F. Goethals, M. D. Backer, R. Haesen, J. Vandenbulcke, and G. Dedene, "Coordinating COTS applications via a business event layer," *IEEE Software*, vol. 22, no. 4, pp. 28–35, 2005.

[35] 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.

[36] H. Mannaert, J. Verelst, and K. Ven, "Exploring concepts for deterministic software engineering: Service interfaces, pattern expansion, and stability," in *Proc. IEEE Intl. Conf. on Software Engineering Advances (ICSEA)*, 2007.

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