An Explorative Study of Module Coupling and Hidden Dependencies based on the Normalized Systems Framework

Dirk van der Linden, Peter De Bruyn, Herwig Mannaert, and Jan Verelst
University of Antwerp
Antwerp, Belgium
dirk.vanderlinden, peter.debruyn, herwig.mannaert, jan.verelst@uantwerpen.be

***Abstract*—Achieving the property of evolvability is considered a major challenge of the current generation of large, compact, powerful, and complex systems. An important facilitator to attain evolvability is the concept of modularity: the decomposition of a system into a set of collaborating subsystems. As such, the implementation details of the functionality in a module is hidden, and reduces complexity from the point of view of the user. However, some information should not be hidden if they hinder the (re)use of the module when the environment changes. More concretely, all collaborating modules must be available for each other. The way how a collaborating module is accessible is also called module coupling. In this paper, we examined a list of classifications of types of module couplings. In addition, we made a study on the implications of the used address space for both data and functional constructs, and the implications of how data is passed between modules in a local or remote address space. Several possibilities are evaluated based on the Normalized Systems Theory. Guidelines are derived to improve reusability.**

Keywords-Reusability, Evolvability, Modularity, Coupling, Address space.

I. INTRODUCTION

Modern technologies provide us the capabilities to build large, compact, powerful, and complex systems. Without any doubt, one of the major key points is the concept of modularity. Systems are built as structured aggregations of lower-level subsystems, each of which have precisely defined interfaces and characteristics. In hardware for instance, a USB memory stick can be considered a module. The user of the memory stick only needs to know its interface, not its internal details, in order to connect it to a computer. In software, balancing between the desire for information hiding and the risk of introducing undesired hidden dependencies is often not straightforward. However, these undesired hidden dependencies should be made explicit [1]. Experience contributes in learning how to deal with this issue. In other words, best practices are rather derived from heuristic knowledge than based on a clear, unambiguous theory.

Normalized Systems Theory has recently been proposed [2] to contribute in translating this heuristic knowledge into explicit design rules for modularity. In this paper, we want to evaluate which information hiding is desired and which is not with regard to the theorems of Normalized Systems. The Normalized Systems theorems are fundamental, but it is not always straightforward to check implementations in different application domains against these theorems. This paper aims at deriving more concrete guidelines for software development in a PLC environment on a conceptual level.

Doug McIlroy already called for *families of routines to be constructed on rational principles so that families fit together as building blocks. In short, [the user] should be able safely to regard components as black boxes* [3]. Decades after the publication of this vision, we have black boxes, but it is still difficult to guarantee that users can use them safely. However, we believe that a lot of necessary knowledge to achieve important parts of this goal are available and we should primarily document all the necessary unambiguous rules to make this (partly tacit) knowledge explicit.

In this paper, we examined a list of classifications of types of module couplings, and evaluated in which terms these types are contributing towards potentially compliance with the Normalized Systems theory. These couplings are studied in an abstract environment [1]. Further, we extended this study by placing the constructs in an address space, and evaluated the consequences. This evaluation is based on some case studies in an IEC 61131-3 programming environment by way of small pieces of code [4]. We investigated on how different data constructs relate to a local or a remote memory address space, and which consequences these relations have to functional modules. Next, we placed the focus on the functional constructs and paradigms, which also reside in a local address space and might have a coupling to a remote address space. We investigated the potential to use them complying the Normalized Systems principles. Finally, we present an set of derived, more concrete principles.

The paper is structured as follows. In Section II, the Normalized Systems theory will be discussed. In Section III, we discuss categories of coupling, seen in an abstract way. In Section IV, we give an overview of how data can be passed between functional modules in a local data memory address space, or coupled with constructs in a remote address spaces. In Section V, we focus on constructs for functionality, and how they can be coupled (locally or remotely). A summary of the evaluations and guidelines is given in Section VI. Finally, Section VII concludes the paper.

II. Normalized Systems

The current generation of systems faces many challenges, but arguable the most important one is evolvability [5]. The evolvability issue of a system is the result of the existence of Lehman's Law of Increasing Complexity which states: "As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it" ([6] p. 1068). Starting from the concept of systems theoretic stability, the Normalized Systems theory is developed to contribute towards building systems, which are immune against Lehman's Law.

A. Stability

The postulate of Normalized Systems 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).

Consequently, the impact of a change should only depend on the nature of the change itself. Systems, built following this rule can be called stable systems. In the opposite case, changes causing impacts that are dependent on the size of the system, are called *combinatorial effects*. To attain stability, these combinatorial effects should be removed from the system. Systems that exhibit stability are defined as *Normalized Systems*. Stability can be seen as the requirement of a linear relation between the cumulative changes and the growing size of the system over time. Combinatorial effects or instabilities cause this relation to become exponential (Figure 1). The design theorems of Normalized Systems Theory contribute to the long term goal of keeping this relation linear for an unlimited period of time, and an unlimited amount of anticipated changes to the system.

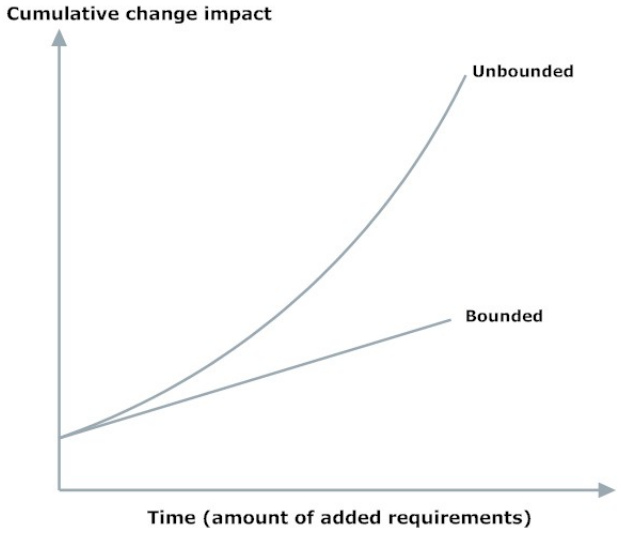

Figure 1. Cumulative impact over time

B. Design Theorems of Normalized Systems

In this section, we give an overview of the design theorems or principles of Normalized Systems theory, i.e., to design systems that are stable with respect to a defined set of anticipated changes:

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

Please note that these changes are associated with software primitives in their most elementary form. Hence, real-life changes or changes with regard to 'high-level requirements' should be converted to these elementary anticipated changes [7]. We were able to convert all real-life changes in several case studies to one or more of these abstract anticipated changes [8][9]. However, the systematic transformation of real-life requirements to the elementary anticipated changes is outside the scope of this paper. In order to obtain systems theoretic stability in the design during the implementation of software primitives, Normalized Systems theory prescribes the following four theorems:

1) Separation of concerns:

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

This theorem focuses on how tasks are structured within processing functions. Each set of functionality, which is expected to evolve or change independently, is defined as a change driver. Change drivers are introducing anticipated changes into the system over time. The identification of a task should be based on these change drivers. A single change driver corresponds to a single *concern* in the application.

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 theorem focuses on how data structures are passed to processing functions. Data structures or *data entities* need to be able to have multiple versions, without affecting the processing functions that use them. In other words, data entities having the property of data version transparency, can evolve without requiring a change of the interface of the action entities, which are consuming or producing them.

3) Action version transparency:

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

This theorem focuses on how processing functions are called by other processing functions. Action entities need to be able to have multiple versions without affecting any of the other action entities that call them. In other words, action entities having the property of action version transparency, can evolve without requiring a change of one or more action entities, which are connected to them.

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

This theorem focuses on how calls between processing functions are handled. Coupling between modules, that is due to errors or exceptions, should be removed from the system to attain stability. This kind of coupling can be removed by exhibiting state keeping. The (error) state should be kept in a separate data entity.

III. Evaluation of types of coupling

Coupling is a measure for the dependencies between modules. Good design is associated with low coupling and high reusability. However, merely lowering the coupling is not sufficient to guarantee reusability. Classifications of types of coupling were proposed in the context of structured design and computer science [10][11]. The key question of this paper is whether a hidden dependency and, therefore, coupling is affecting the reusability of a module? In general, the Normalized Systems theorems identify places in the software architecture where high (technical) coupling is threatening evolvability [12]. More specifically, we will focus in this section on several kinds of coupling and evaluate which of them is lowering or improving reusability. The sequence of the subsections is chosen from the most tight type coupling to the most loose type of coupling.

A. Content coupling

Content coupling occurs when module A refers directly to the content of module B. More specifically, this means that module A changes instructions or data of module B. When module A branches to instructions of module B, this is also considered as content coupling.

It is trivial that direct references between (internal data or program memory of) modules prevent them from being reused separately. In terms of Normalized Systems, content coupling is a violation of the first theorem, separation of concerns. Achieving version transparency is practical not possible. The same can be said about separation of states.

This intent to avoid content coupling is not new, other rules than those of the Normalized Systems already made this clear. For instance, Dijkstra suggested decades ago to abolish the goto statement from all 'higher level' programming languages [13]. The goto statement could indeed be used for making a direct reference to a line of code in another module. Together with restricting access to the memory space of other modules, Dijkstra's suggestion contributed to exile content coupling out of most modern programming languages. Note that in the IEC 61131-3 standard, the Instruction List (IL) language still contains the JMP (jump) instruction. For this and other reasons, IL is considered a low level language, and similar to assembly.

B. Common coupling

Common coupling occurs when modules *communicate using global variables*. A global variable is accessible by all modules in the system, because they have a memory address in the 'global' address space of the system. If a developer wants to reuse a module, analyzing the code of the module to determine which global variables are used is needed. In other words, a white box view is required. Consequently, black box use is not possible. In terms of Normalized Systems, common coupling is a violation of the first theorem, separation of concerns.

We add however, that not the *existence* but the *way of use* of global variables violates the separation of concerns theorem. A global variable is in fact just a variable in the scope of the main program. When these *global variables* are treated like a kind of *local variables in the scope of the main program*, they do not cause combinatorial effects. However, when these variables are passed to the submodular level without using the interface of (sub)modules, which are called by the main program, they can cause combinatorial effects. Since the use of global variables in case of common coupling is not visible through the (sub)module's interface, this way to use these global variables is considered to be a hidden dependency. And since common coupling is a violation of separation of concerns, this is an undesired hidden dependency with respect to the safe use of black boxes.

As a research case, we used global variables in a proof of principle with IEC 61131-3 code, which complies with Normalized Systems [9]. The existence of global variables was needed for other reasons than mutual communication between modules (i.e., connections with process hardware). In this project, the global variables were passed via an interface from one module to the other. In some cases, having a self-explaining interface between collaborating modules is enough to comply with the separation of concerns principle. In other cases, dedicated modules called connection entities are needed to guarantee this separation. In this paper, we investigated in which cases there is a need for a connection entity or not (see following subsections).

C. External coupling

External coupling occurs when two or more modules communicate by using an external (third party?) database, communication protocol, device or hardware interface. The external entity, system or subsystem is accessible by all (internal) modules. Consequently, the support (e.g., fault handling) for the external access has to be included for all modules.

Support for this particular external access is a concern. Every module also includes at least one core functionality, which is also a concern. Having more than one concern in a single module is a violation of the separation of concerns principle. Indeed, when the external entity receives

an update, every module, which is calling the external entity, needs an update too. This is an example of a combinatorial effect.

To avoid this kind of combinatorial effect, one should dedicate a special module - a connection entity - to make the link with the external technology. More precisely, one connection entity for every version or alternative external technology. Version tags can be used to select the appropriate connection entity. Each internal module should call the connection entity to map parameters with the external entity.

Such a connection entity is considered to be a supporting task. Separating the core task from supporting task does not have to decrease cohesion. On the contrary they can nicely fit together on the next modular level. In other words, the core task module can be 'hosted' together with one or more supporting task module in a higher-lever module.

D. Control coupling

Control coupling occurs when module A influences the execution of module B by passing data (parameters). Commonly, such parameters are called '*flags*'. Whether a module with such a flag can be used as a black box depends on the fact whether the interface is explaining sufficiently the meaning of this flag for use. If a white box view is necessary to determine how to use the flag, black box use is not possible. The evaluation of control coupling in terms of reusability is twofold. On the one hand, adding a flag can introduce a slightly different functionality and improve the reuse potential. For example, if a control module of a motor is supposed to control pumping until a level switch is reached, a flag can provide the flexibility to use both a positive level switch signal and an inverted one (i.e., positive versus negative logic). On the other hand, extending this approach to highly generic functions, would lead in its ultimate form to a single function *doIt*, that would implement all conceivable functionality, and select the appropriate functionality based on arguments. Obviously, the latter would not hit the spot of reusability.

One of the key questions during the evaluation of control coupling is: how many functionalities should be hosted in one module? In terms of Normalized Systems, the principle 'separation of concerns' should not be violated. The concept of change drivers brings clarity here. A module should contain only one core task, eventually surrounded by supporting tasks. Control coupling can help to realize theorem 2 (data version transparency) and theorem 3 (action version transparency) by way of version selection. The calling action is able to select a version of the called action based on control coupling. We conclude that *control coupling should be used for version selection only*.

Control coupling, as a way of connecting two or more modules, says something about the functional impact of the coupling, not about how the coupling is realized. Consequently, control coupling does not influence the choice whether a connection entity is necessary or not.

E. Data coupling

Data coupling occurs when two modules pass data using simple data types (no data structures), and every parameter is used in both modules.

Realizing theorem 3 (action version transparency) is not straightforward with data coupling, since the introduction of a new parameter affects the interface of the module. This newer version of the interface could not be suitable for previous action versions, and could consequently not be called a version transparent update. Not all programming languages support flexibility in terms of the amount of individual parameters. Changing the datatype, or removing a parameter is even worse.

Note that the disadvantage of data coupling, affecting the module's interface in case of a change, does not apply on reusing modules, which are not evolving. This can be the case when working with system functions, e.g., aggregated in a system function library. However, problems can occur when the library is updated. We will give more details about this issue in the next section.

When working with separated, simple data types as a set of parameters, every change requires a change of the interface of the module. Since we do not consider 'changing the interface' as one of our anticipated changes, this should be avoided. Huang et al. emphasized that it is important to separate the version management of components with their interfaces [14]. As such, the interface can be seen as a concern, and should consequently be separated to comply with the separation of concerns principle.

In other words, in case the development environment does not support a flexible interface for its modules, data coupling can cause combinatorial effects. In case mandatory arguments are removed in a new version, a flexible development cannot guarantee the absence of combinatorial effects.

F. Stamp coupling

Stamp coupling occurs when module A calls module B by passing a data structure as a parameter when module B does not require all the fields in the data structure.

It could be argued that using a data structure limits the reuse to other systems where this data structure exists, whereas only sending the required variables separately (like with data coupling) does not impose this constraint. However, we emphasize that the key point of this paper does not concern *reuse* in general. Rather, it focuses on *safe reuse* specifically. Stamp coupling is an acceptable form of coupling. With regard to the first theorem, separation of concerns, one should keep the parameter set (data entity), the functionality of the module (action entity) and the interface separated. Keeping the interface unaffected, while the data entity and action entity are changing, can be realized with stamp coupling. Note that stamp coupling should be

combined with the rule that fields of a data structure can be added, but not modified or deleted. This rule is necessary to enable version transparency.

Note that if the data structure in a stamp coupling scenario increases, it becomes convenient to pass the structure by reference (see Section IV-D). As such, memory use and copying processes can be limited. However, referring to the data structure requires the stamp coupling to be applied between modules which reside in the same address space (see Section V-D).

G. Message coupling

Message coupling occurs when communication between two or more modules is done via message passing. With message passing, a copy of a data entity is sent to a so-called communication endpoint. An underlying network does the transport of (the copy of) the data entity. This underlying network can offer incoming data, which can be read via the communication endpoint. Message passing systems have been called 'shared nothing' systems because the message passing abstraction hides underlying state changes that may be used in the implementation of the transport.

The property 'sharing nothing' makes message coupling a very good incarnation of the separation of concerns principle. Please note that asynchronous message passing is highly preferable above synchronous message passing, which violates the separation of states principle. The system works with copies of the data, and the states of the transport are separated from the application which is producing or consuming the data. This concept complies with the separation of states principle.

In comparison with stamp coupling, stamp coupling can be realized by passing a pointer, which refers to the data structure. To implement this, both modules should share the memory address space, where the pointer is referring to. Since the concept of message coupling does not share anything, also no address space, every data passing works with copies. For this reason, message coupling is considered the most loosely coupled of all categories.

Message coupling implies additional functionality with regard to the modules which need to exchange data. To comply with the separation of concerns principle, this additional functionality should be separated from the core functionality of the collaborating modules. Consequently, while the data structure in a stamp coupling scenario – in a common address space – can be used directly by the collaborating modules, at least two connection entities are required when these modules reside in a different address space (see Section V-D)).

H. Summary of the theoretic evaluation of couplings

The existing categorization of coupling is based and ordered on how tight or how loose the discussed coupling type is. We agree that in general loose coupling is better than tight coupling, but there are more important consequences based on the different types of coupling. It is not too surprising that, following our evaluation, we discourage the use of the two most tight types of coupling, i.e., content coupling and common coupling. However, other conclusions are not based on how tight a type of coupling is. For example, control coupling is a special one, because it is the only discussed type which says something about the functionality of the connected modules. All other types says something about how these modules are coupled. Data coupling and stamp coupling are alternatives for each other, while other types can be used complementary. We highly recommend stamp coupling in stead of data coupling, because data coupling can cause combinatorial effects.

Stamp coupling can be combined with control coupling, message coupling or partly external coupling (depending on the application). Control coupling should be used for version selection only. Stamp coupling can be used as it is in cases where the collaborating modules reside in the same system. In case these collaborating modules reside in different systems, stamp coupling has to be combined with message coupling. In case the collaboration includes external entities, from which we cannot control the evolution, connection entities are necessary, which is a prerequisite to use external coupling without potentially causing combinatorial effects.

IV. Data memory address space and its borders

The discussion about message coupling illustrates that a reference to a variable in a particular address space can be seen as an occurrence of a hidden dependency. In this section, we investigate this more in depth, and discuss several software constructs which have a relation with one or more memory address spaces.

In its most elementary form, programs are nothing but a sequence of instructions, which perform operations on one or more variables. These variables correspond to registers in the data memory of the controller, and the instructions correspond to registers in the program memory. The instructions are executed in sequential order, but instructions for selections and jumping to other instructions are available. In this elementary kind of programs, there is no explicit modularity at all, any instruction can read any variable in the program, and jumping from any instruction to any other instruction is possible. For this purpose, we had in the early ages of software development an instruction, which has become well-known: the goto-statement. Dijkstra called for the removal of the goto-statement in higher level languages [13], and this call is mainly addressed. However, the JMP (jump) instruction is still available in the lower level language Instruction List (IL) of the IEC 61131-3 standard for PLC (Programmable Logic Controller) programming [4]. Also, in surprisingly recent literature, goto elimination is still a research objective [15].

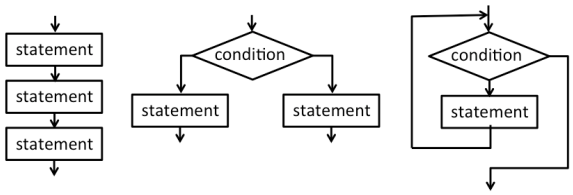

Figure 2. Concatenation, Selection, and Iteration

Alternatively, Dijkstra elaborated on the concepts concatenation, selection and iteration (Figure 2) to bring more structure in a program [16]. However, these concepts do not force modularity. In terms of Normalized Systems theory reasoning, the separation of concerns principle is not addressed. Because of the lack of clearly identifiable modules, the other theorems cannot be evaluated as well.

In this section, we discuss an amount of software constructs and how they relate to the address space, and whether the desired coupling has to cross the borders of this address space. We evaluate some concepts or paradigms based on the Normalized Systems theorems. We start our discussion with the very first attempt to build modular software systems: the 'closed subroutines' of Wilkes et al. (1959). Next, we discuss the concept of data variables, and how their scope can differ corresponding their definition. Further, we discuss variables which can be exchanged between modules. These kind of variables are typically called parameters or arguments. Two main ways how they can be passed is 'by value' or 'by reference', which will be discussed. Finally, the concepts of static and external variables will be discussed.

A. Subroutines

Wilkes et al. introduced the concept of subroutines, which they termed a closed subroutine [17]. The concept of subroutines is the first form of modularity. A subroutine, also termed subprogram, is a part of source code within a larger program that performs a specific task. As the name subprogram suggests, a subroutine can be seen as a piece of functionality, which behaves as one step in a larger program or another subprogram. A subroutine can be called several times and/or from several places during one execution of the program (including from other subroutines), and then return to the next instruction after the call once the subroutine's task is done (Figure 3).

Dijkstra reviewed the concept of subroutines in [16]. Following this review, the concept of subroutines served as the basis for a library of standard routines, which can be seen as a nice device for the reduction of program length. However, the whole program as such remained conceived as acting in a single homogeneous store, in an unstructured state space; the whole computation remained conceived a single sequential process performed by a single processor ([16], p. 46). In other words, the subroutine shares its data

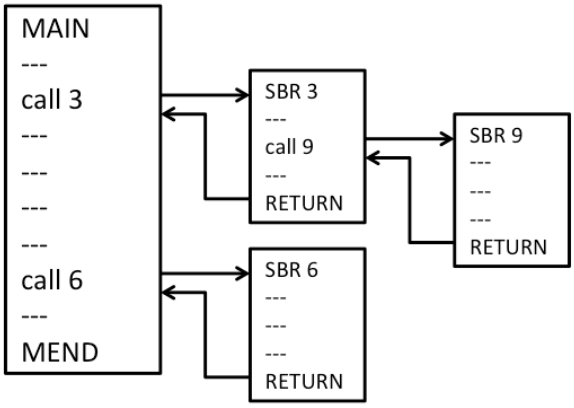

Figure 3. Subroutines

memory address space with the main program and other subroutines (if these exist). The return address of a closed subroutine can not be seen as a parameter. Rather, it looks like a well-placed jump.

In terms of Normalized Systems, progress is made towards the separation of concerns principle, but it is not fully addressed yet. Indeed, the details of the functionality in a subroutine is separated from the main program (which can be seen as a desired hiding of information for the reader of the main program), but the data of the subroutine is not. In fact, the lack of a local data memory address space in a 'closed subroutine' implies a violation of the separation of concerns principle. On the side of functionality the concerns 'main program' and 'closed subroutine' are separated, but on the side of data these concerns are not separated. Because of the lack of separation of data memory address space, the separation of states principle cannot be met. The separation of states principle implies the buffering of every call to another module. As such, when the called module does not respond like expected, the calling module can handle the unexpected result based on the buffered state. In other words, every module needs its own local memory to store its state.

B. Variables

A variable is a storage location and an associated symbolic name, which contains a value. Note that this concept is very explicit exemplified in contemporary Simatic S7 PLCs, where the programmer can choose for usage of *absolute addresses* and *symbolic addresses* [18]. In this specific environment, the programmer has to manage the data memory address space. For computer scientists, this might look old-fashioned, but for contemporary PLC programmers this is an important subject. Moreover, data memory address space cross references are tools which are commonly used to heuristically prevent combinatorial effects caused by common coupling. More general, the variable name is the usual way to reference the stored value, and a compiler

is doing the data memory allocation and management by replacing variables' symbolic names with the actual data memory addresses at the moment of compilation. The use of abstract variables in a source code, which are replaced by real memory during compilation is undoubtedly an improvement for reusability of the source code. However, when the memory is still shared throughout the whole system, these variables are called *global* variables, and require a name space management to prevent name conflicts. In other words, the problem of potential address conflicts is moved to potential name conflicts. In terms of Normalized Systems, when modules need global variables to exchange data, this is not really an improvement in relation to the concept of closed subroutines of Wilkes et al. ([17]).

A group of research computer scientists abandoned the term 'closed subroutine' and called modules 'procedures' in the ALGOL 60 initiative [19]. The main novelty was the concept of *local* variables. In terms of memory address space, the concept 'scope' was introduced, i.e., the idea that not all variables of a procedure are homogeneously accessible all through the program: local variables of a procedure are inaccessible from outside the procedure body, because outside they are irrelevant. What local variables of a procedure need to do in their private task is its private concern; it is no concern of the calling program [16]. In terms of Normalized Systems, local variables contribute in addressing the separation of concerns principle. A point of potential common coupling is still the fact that global variables –which are declared outside the module– are still accessible from the inside of the module. When these global variables are used in the module, without documenting this for the user, we have a violation of the separation of concerns principle. The use of undocumented and thus invisible or hidden global variables in a module makes it impossible to evaluate compliance with the Normalized Systems theorems. In other words, code analyses or white box inspection is needed to decide whether the module can be (re)used in a specific memory environment. Providing a list of the used global variables in the module documentation would be an improvement, but passing the global variables to the module as parameters or arguments is even better. The reason why this is better, is because of a better separation of the local and global address space.

C. Parameters and arguments

Having a local data memory address space contributes in separating concerns, but since the aim of software programs is generally performing operations on data entities, we should be able to exchange data between these separated memory address spaces. The question is: how should this be done? In principle, there are two possible approaches: or we exchange data by way of global variables, or we use a modular interface, which consists of input- and output parameters or arguments.

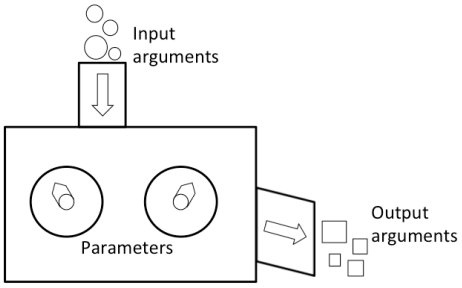

Figure 4. Function machine with parameters and arguments [20]

The terms parameter and argument are sometimes easily used interchangeably. Nevertheless, there is a difference. We use the function machine metaphor to discuss how functionality can depend on parameters (Figure 4) [20]. The influence of parameters should be seen as a configuration of the functionality, while the arguments are, following this metaphor, the material flow. This can also be exemplified with a proportional-integral-derivative (PID) controller. A PID controller calculates an 'error' value as the difference between a measured process variable and a desired setpoint. The controller attempts to minimize the error by adjusting the process control inputs. The proportional, the integral and derivative values, denoted P, I, and D, are parameters, while the measured process value and the setpoint are the arguments.

From a software technical point of view, it is not important to treat parameters and arguments different when these values are exchanged between modules. However, from an application point of view, they should be aggregated differently. Like discussed in Section II, the functionality and data should be encapsulated as action entities and data entities, respectively. Since it is imaginable that the configuration of functionality (parameters) changes independently of a potential change of, e.g., the data type of the arguments, these data constructs should be separated following the separation of concerns principle. Also, the action entities, which manipulate configuration data entities, should be separated from action entities, which manipulate process data entities. Besides, the user access rights might be different, e.g., adjusting the configuration should be done by maintenance engineers, while process data might be manipulated by system operators. For simplicity reasons, in what follows, we use the term 'data passing' for both cases, in the assumption that the manipulation of arguments and parameters is separated in different modules. These separated submodules should collaborate based on stamp coupling. In its simplest form, the data structures which can be used for stamp coupling are called structs, records, tuples, or compound data. Conceptually, such data structures have a name and several data fields. In the next section, data objects

will be discussed.

To come back on our discussion about module dependencies, data passing can be based on a shared data memory address space between the calling and the called module (i.e., via global variables), or on the module's interface (i.e., via in/out variables). When we put ourselves into the position of a software engineer, who want to reuse a module, both the module and the definition of the global variables should be copied before the module can be reused. More specifically, to not create unused global variables in the target system (or to minimize potential name conflicts), the software engineer should only copy the global variable definitions, which are used in the module. It is imaginable that this is not in all situations straightforward, unless we provide a list or declaration of all used global variables as a documentation of the module. When the software engineer, into the process of module evolution, considers to change the module, any change on one or more of the used global variables, requires a corresponding change in the global variable definitions of the system. In case the global variables are also used in other modules, the need to perform a corresponding change in each of these modules is an occurrence of a combinatorial effect. In terms of Normalized Systems, passing data by way of global variables (common coupling) is a violation of the separation of concerns principle. Adding a global variable could be deemed to comply with the version transparency theorems, but this could be not so convenient if more engineers are working on the same project, and the chance on naming conflicts increases compared to the potential addition of a local variable.

To prevent these disadvantages, passing data by way of in/out variables, i.e., the module's interface, is more convenient and increases maintainability. The module as a construct is a way to separate the address space of the module with the address space of the 'outside', and the module's interface performs the function of a managed gateway for data passing. The reusability of the module is improved when strictly using local variables or in/out variables. However, other dependencies are still a point of interest, which will be discussed in the next section.

D. Pass by value or by reference?

Data passing *by value* means that an input variable is copied to an internal register of the module, and return *by value* means that a produced value is stored in an internal register, and copied to an output variable at the end of the processed functionality. In contrast, passing and return *by reference* means that the in/out variable is stored in a memory space outside the module, while only a reference or address to this memory space is used in the module. The in/out variable is never copied because the link with the memory outside the module remains available during the processing of the functionality.

It is not too surprising that, following our evaluation, data passing by value is isolating and separating the inside of the module better from the outside than if the same set of in/out variables would be passed by reference. In other words, in the case of pass by reference, the memory address space, which is surrounding the module, is a dependency of the module. To eliminate combinatorial effects, any dependency needs some attention. However, in this case, the dependency of memory address space is not necessarily causing combinatorial effects. In case the coupled modules reside in the same memory address space, passing parameters by reference does not cause combinatorial effects. In other words, one must make sure that the coupling is not crossing the borders of the memory address space of the considered system, which is 'hosting' the coupled modules. In case the coupling is crossing the borders of the memory address space, it has to be combined with message coupling, which implies data passing by value.

In an IEC 61131-3 environment, the length of arrays and strings are explicitly defined. This is safer in comparison with systems where this length is flexible at runtime. Note that a 'by reference' in/out variable is a pointer to the start memory address of a variable. When there is flexibility about the end address of this memory variable —e.g., an array with no explicit defined length— the pointer+index might refer to an address outside the scope of the intended variable. There is a risk that this situation becomes similar to content coupling. However, a lot of software systems tackle this problem by means of exception handling.

When we evaluate the choice between 'pass by value' or 'pass by reference' based on the Normalized System theorems, 'pass by value' contributes better towards the separation of concerns principle, by copying in-variables from the 'outside' to internal registers, and copying internal registers to out-variables after processing the functionality. In/out variables which are passed by reference always maintain a reference in the external address space, which can be seen as a dependency. Since this type of dependency can be automatically managed for every individual variable by the compiler —by way of memory (re)mapping during compilation— we do not call this dependency a violation of the separation of concerns principle from the point of view of the application software engineer. However, the approach has its limitations.

Kuhl and Fay emphasized that a static reconfiguration, which requires a complete shutdown of a system, is more costly than a dynamic reconfiguration, which can be performed without a complete shutdown [21]. Since we do not have control about how a compiler is doing the memory (re)mapping of (the reference address of) in/out variables which are passed by reference, we should assume that a dynamic configuration is limited by the data memory address space. More specifically, when a change is introduced in a module which processes in/out variables by reference, a

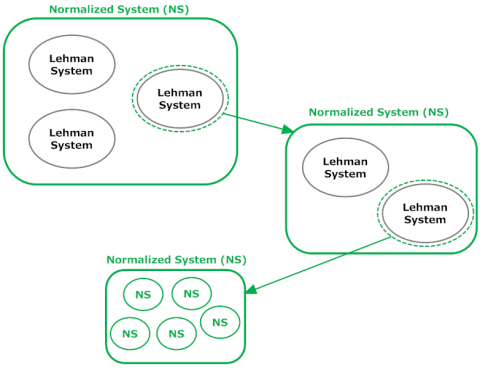

Figure 5. Different levels of modularity [22]

memory remapping of the surrounding system is necessary, and thus requires a shutdown of this system.

It is important that the application engineer is aware of this discussed limitation, especially when the choice has to be made to pass by reference or not. One should be aware that copying pass-by-value-variables costs processor time and memory space (which can be even more than strictly required when applying stamp coupling). Remember that the Normalized Systems authors advocate a higher granularity, i.e., smaller modules with the consequence that – for the same functionality– the amount of modules increases, including the (amount of) modular interfaces.

The definition of the theorem 'separation of concerns' has a focus on separation of 'tasks' (Section II), which might be interpreted as a separation of functionality. However, a concern can also be interpreted as a data memory address space, let it be on a different level of aggregation. More specifically, separation of functionality is advantageously on the lowest level of modularity, —decisions are supported with the concept of change drivers— but on a higher level the technical environment, e.g., the data memory address space, might be considered a concern. In other words, we propose that higher level constructs (aggregating one or more entities) can use the concept of passing by reference internally to let entities communicate mutually by way of stamp coupling, reusing the same interface for every entity. This might limit the consequences of the higher granularity by enabling the reuse of modular interfaces. More levels of this design might be possible in cascade, like suggested in the migration scenario's in Figure 5 [22].

E. Static and external variables

In his thinking on the recursive procedure, Dijkstra praised the concept of local variables, but he also mentioned the shortcoming of life-time of local variables. Local variables are 'created' upon procedure entry, and cease to exist when the procedure ends. The fact that local variables relate to an instantiation and only exist during that specific instantiation makes it impossible for the procedure to transmit information behind the scenes from one instantiation to the next ([16], p. 48). In this paper, we do not wish to advocate recursive procedures, but we do emphasize that the concept of static local variables (i.e., local variables which can remember their state of the previous run or incarnation) is advantageous towards the separation of states principle. The term *static* refers to the fact that the memory for these variables are allocated statically –at compile time– in contrast to the local variables, whose memory is allocated and deallocated during runtime. This concept is clearly exemplified in [18], where local (temporal) variables in a module of the form FC (Function) cannot remember their previous state, and local (static) variables in a module of the form FB (Function Block) can. For storing static variables, this type of PLCs use dedicated data memory constructs they call Data Blocks (DBs). In the case they connect such a DB to an FB they call it an *instance* DB.

The concept of external variables requires some explanation concerning definition and declaration. The *definition* of global variables decides in which memory address space they can be used, and the *declaration* of these global variables in the documentation of a module informs the potential user of the module that these global variables are needed to be able to use the module. The definition of a variable triggers the compiler to allocate memory for that variable and possibly also initializes its contents to some value. A declaration however, tells the compiler that the variable should be defined elsewhere, which the compiler should check. In the case of a declaration there is no need for memory allocation, because this is done elsewhere. The VAR_EXTERNAL keyword in an IEC 61131-3 environment indicates that the following variable is declared for the module where this keyword is used, and defined elsewhere (probably global).

Unfortunately, following a study of de Sousa, the details of defining global variables and declaring external variables are discussable to the letter of the IEC 61131-3 standard [23]. This author even doubt whether it is advantageous to have the possibility of external variable declarations within function block declarations, because passing a global variable via the keyword VAR_IN_OUT has a similar effect. In earlier work, we also advocated the use of in/out variables in an IEC 61131-3 project [9], but still, when we evaluate the concept of external variables based on the Normalized Systems theory, the explicit declaration of the use of global variables in a module eliminates potential combinatorial effects caused by common coupling. In this context, it is interesting that de Sousa considered VAR_EXTERNAL variables as belonging to the interface ([23] p. 317).

V. CONSTRUCTS FOR FUNCTIONALITY

In the previous section, we discussed mainly the concerns of data memory, and also how data memory relates to the first type of software modules, 'closed subroutines',

and its successor 'procedures'. The latter can have local variables, and an interface. The modular interface consists of a name for the procedure, and the input and output data variables, which are preferably data structures. We now discuss some other types of modules, which can be considered as extensions of the concept of the procedure and its interface.

A. Object-Oriented programming

The main new construct for implementing modules in object-oriented languages is the class. A class consists of both data variables (member variables) and functionality (methods). Methods can have their own local variables, but can also access the member variables and other methods of the class it belongs to. To allocate data memory and enable the methods to really work, a class needs to be instantiated or constructed to make an object. Objects of the same class can co-exist. Data and functionality are tightly coupled in an instance (object). Methods which are declared as public, are visible for other objects. Memory variables are normally considered as private to the class and, therefore, invisible for other objects. The interface of a method consists of a name for the method, and input and output variables. An object-oriented design consists of a network of objects calling methods of other objects, which can be implemented as data coupling or stamp coupling.

Since each method has its own interface, and a class can contain multiple methods, an object as a module can have multiple interfaces. Classes can be extended with the concept of inheritance. This concept envisaged to mimic the concept of ontological refinement. Just like a bird is a special type of animal, and a sparrow a special type of bird, inheritance was created to define classes as refinements of other classes. Such a subclass would inherit the member variables and methods of a superclass, and extend it. However, Mannaert and Verelst state that in practice, very few programming classes are in line with the assumption that object-oriented inheritance is based on ontological refinements ([2], p. 29). If we cannot count on ontological refinements, a class can also be seen as just an amount of methods, grouped together based on the intuition of the programmer, and sharing the same set of member variables. When the size of such a class grows, the situation becomes comparable with a system based on procedures, having their own local variables, but sharing the system's global variables.

In terms of Normalized Systems, we evaluate that the object-oriented programming paradigm is not guaranteeing compliance with the separation of concerns principle. First, in case the data type or data representation can change independently from the functionality, the tight coupling between data and functionality makes version transparency not straightforward. For example, consider that in an application, a house-number-field changes its data type from numeric to alpha-numeric, without any functional change. The datatype change might require the functionality to change, too. As such, it seems possible that combinatorial effects occur, which makes version transparency infeasible when the size of a system grows. Second, when the size of a class grows, the member variables are similar with (class-wide) global variables. Consequently, common coupling between methods is imaginable and combinatorial effects can occur. As a remedy, this dependency could be made explicit by declaring the use of every member variable in a method by way of declaration concept similar to the the declaration of external variables. Indeed, from the point of view of a method, a class member variable can be seen as 'external'.

Public methods can be called via their interface, as if they make part of the programming environment. However, they belong to a class. If someone wants to reuse such a method in another system, at least the 'hosting' class should be copied as well. In addition, other classes which contain coupled methods should be copied, too (note that a class can contain methods, from which the code include the construction of objects, based on other classes). In other words, public methods, which reside in classes, are available in a flat name space. Any public method can call any other public method, which can result in a complex network of calling and called method, residing in the same or different objects. In an evolving system, the required version management between the calling and called (public) methods (with additionally tightly coupled data), is not straightforward. To be able to keep track of all couplings, including the versions of these methods, we propose a similar explicitation like we did for memory variables. The method interface should include a declaration or documentation part, which informs the user of all methods which are called inside the method, including the object and class version to which they belong. This declaration might be done in a similar way as the declaration of external variables, i.e., the announcement that one or more functional constructs are used or called in the code of the concerning method. In terms of Normalized Systems, we evaluate that methods and classes might comply with the separation of concerns principle, but extra constraints are necessary. There should be only one 'core'-method containing the core functionality of the class, surrounded by supporting methods like cross-cutting concerns. Also version transparency should be an extra constraint when using the object oriented paradigm.

The concept of inheritance does not guarantee version transparency, because it is based on an anthropomorphical assumption, which is not realistic in all cases. It would be better to implement explicit version management, based on version IDs. This version management should be twofold: first, the versions of data memory entities (including type or representation) should be made explicit, and second, the versions of the functionality, how the versions of data memory entities relate to the versions of functionality and vice versa should be made explicit as well.

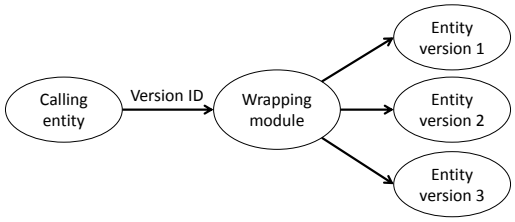

Figure 6. The concept of version wrapping

We do discuss some potential drawbacks of the object-oriented paradigm, but we emphasize that it is possible to build evolvable systems, based on the object-oriented paradigm, complying the Normalized Systems theorems. However, the object oriented paradigm itself does not guarantee the property of evolvability. Additional constraints are necessary to eliminate combinatorial effects. One of the key remarks is that an object should not contain more than one core functionality, and functionality should be separated from data representation. One of the possibilities is the introduction of data objects and functional objects. In addition, the use of memory variables and methods in a method should be declared on a similar way like the concept of external variables. We also think that polymorphism, combined with explicit version management might be an alternative for inheritance. This alternative could exhibit version transparency, but more elaboration and future work is needed to figure this out.

B. Modules in IEC 61131-3

In an IEC 61131-3 environment, we have Functions (FCs), which have in addition to the input and output variables only temporary local variables. The Function Block (FB) construct can have static local variables, too. More general, these constructs are called Program Organization Units (POU), and are stored in a flat program memory space. On the same level global variables and derived data types are defined (in IEC 61131-3 terms, as a *configuration* definition). Note that, besides the functionality, FBs need data memory before they can actually run. Several FB instances can co-exist with separated data memory. This concept is very similar to the object-oriented paradigm. Indeed, Thramboulidis and Frey state that the Function Block concept has introduced in the industrial automation domain basic concepts of the object oriented paradigm [24]. There is a restriction in the behavior definition of the FB: only one method can be defined. There are no method signatures as in common object oriented languages; actually there is no signature even for this one method defined by the FB body. This method is executed when the FB instance is called [24][4]. Note that the object oriented extension of the FB construct that is under discussion in IEC is not considered in this paper.

Polymorphism is not supported in version two of the IEC 61131-3 standard, nor is inheritance [4]. In a commercial IEC 61131-3 environment, the only way to implement version management is doing this explicitly. In earlier work, we proposed the concepts Transparent Coding and Wrapping Functionality [9]. Transparent coding is defined as the writing of internal code in a module which is not affecting the functionality of previous versions. When Transparent Coding is not possible (e.g., because of conflicting functionality of the versions, or when the combination of the functionality of different versions requires too complex code), Version Wrapping can be applied. Following this principle, different versions of a module co-exist in parallel, and a wrapping module selects the desired version based on the version ID (see Figure 6).

As a reflection with regard to the general object oriented paradigm, it is straightforward to implement only one core functionality in an (IEC 61131-3) FB, because following the analysis of Thramboulidis and Frey only one method is defined in a FB [24]. However, software application engineers tend to extend the possibilities of FBs by way of control coupling. In other words, it is possible to select different functionality based on parameters. In terms of Normalized Systems reasoning, control coupling should be restricted to version selection only. In this way, several versions can co-exist, but still not more than one core functionality resides in one module.

We also reflect on the issue of separation of data and functionality. If we would do this rigorously and strict, we would abandon the use of FBs and stick to the use of FCs only, because FBs can have static variables, and FCs cannot. This also implies that FBs can call other FBs, but FCs cannot call FBs. Indeed, FCs cannot instantiate FBs because they can not allocate the static memory FBs require in syntactical sense. However, we do advocate the use of FBs, because we think it is advantageous to separate technical data, which can be tightly coupled with the functionality, and content data, which has a meaning with regard to the algorithm which is processed in the functionality. For example, to detect the so-called rising or falling edges, e.g., the arriving of a bottle on a filling location, we need to remember the previous state of a sensor. The memory needed to detect these rising or falling edges is a technical matter, of which we might desire to be hidden. In contrast, the information that the event of arrival occurred, is something important for the process algorithm, e.g., to trigger the filling process of the arrived bottle. Another example is the case of the control of a valve, which includes an alarm state. The valve is operational when the feedback sensors (i.e., open or closed sensors) correspond to the output control (i.e., open or closed commands). However, the valve has a mechanical inertia, i.e., it needs some time to open or close, so having a discrepancy between feedback and control is temporary normal. Typically, a timer construct is used to temporary allow a discrepancy, while not entering the alarm state. The data needed for the technical instance of the timer construct is data we call a technical data entity,

which can be hidden and tightly coupled to the module which is performing the alarming algorithm. The result of the decision whether the valve is in the alarm state or not, is related to the control algorithm of the valve, and should be stored in a separated data entity, or more specifically, passed via the modular interface.

C. Libraries and packages

Libraries are collections of compiled modules, which can be shared among various application programs. In an IEC 61131-3 environment, they can also include the definition of the so-called derived data types, i.e., user defined data types, such as structs. Some libraries are called 'standard' libraries, because the content is specified in a standard (this kind of library functionality is also specified in IEC 61131-3). The functionality offered in a standard library is assumed to be widely known, and application engineers should be able to treat them as if they make part of the programming environment. However, in an IEC 61131-3 environment, the details of standard constructs might slightly differ from one brand to another, because this standard allows the so-called implementation-dependent parameters ([4], annex D).

At first sight, the concept of adding 'standard' or other constructs with a reuse potential by way of libraries sounds interesting. Indeed, when the set of shared functionality is small enough, this concept looks great. However, like Dijkstra already recognized back in 1972, one of the important weaknesses in software programs is an underestimation of the specific difficulties of size ([16], p. 2). Remember that the Normalized Systems theory emphasize the importance of separation of concerns. When we interpret a concern as a module or user defined data type, we can count on an unique identification of these constructs into the name-space borders of an individual library or package. However, when these libraries are selected in the library management tool of a programming environment, these constructs end up in a common flat name space. In other words, name space conflicts can occur when constructs of different libraries end up in the same flat module name space.

This might result in a so-called dependency-hell. This is a colloquial term for the frustration of some software users who have installed software packages, which depend on specific versions of other software packages. It involves for example package A needing package B & C, and package B needing package F, while package C is incompatible with package F. Again, when the amount of selected libraries is limited, one could avoid a dependency-hell. However, when constructs are shared between different developers, who perform maintenance activities or make extension of the same application over time, they might use constructs of the same library, but from a different library version. If it is desired that one construct of a library is used from a early version, and another construct of the same library is used from a recent version, it looks impossible to prevent dependency problems in a flat name space. Also, in [18] the modules have a number and a symbol. This number might conflict with existing modules, or with modules from another library.

To come back on the separation of concerns principle, let us interpret a concern as a library. When different libraries are selected in a programming environment, and all constructs of these libraries end up in the same construct name space, we evaluate this as a violation of the separation of concerns principle. This violation is even worse when two versions of the same library would be selected. If the name of the library is not including the version, it might be even impossible to select both. Having functional constructs or data type definitions in a flat name space is similar to common coupling. The use of a library construct in a module should be documented in order to make an evaluation whether the construct can be used in the concerning module or not. The addition of a module, which is using a conflicting name, indicates a bad separation of the constructs available in the used libraries. We derive that using modules from a library should be restricted to standardized functionality and constructs. The designers of the standard should prevent name conflicts in a similar way how keywords are reserved in a programming language. One should avoid to configure library constructs, dedicated for reuse in specific applications, in a flat name space.

As a remedy, constructs belonging to a specific library could be selected on the level of the module, not on the level of the programming environment. This would mean a kind of localization of library constructs. The declaration part of a module could include a library browser, to select a desired functionality or data type from that library. In addition, the version of constructs and libraries should always be included in the declaration part of the module. In this declaration, the 'hosting' library of a construct, accompanied with its version, should be included as a kind of path. As such, it would be even possible to use co-existing versions of a library construct in the same module, because the concerning constructs are well separated.

D. Distributed calling via messages

In an IEC 61131-3 environment or in truly object oriented languages, a module can only call other 'local' modules. Local means that they need to be available within the same program address space. Libraries are deployed locally in the sense that they are compiled and linked into the same program and memory address space. The concept of inter-process communication allows remote calls to a library or system, which is 'hosted' in another program and memory address space. Following a paper of Birrel and Nelson, remote procedure calls (RPC) appear to be a useful paradigm for providing communication across a network between programs written in a high-level language [26]. The idea of RPC is quite simple. When a remote procedure is invoked, the

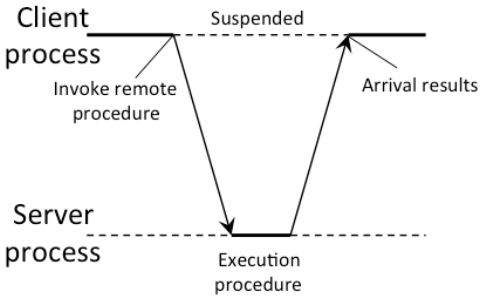

Figure 7. Principle of RPC between client and server [25]

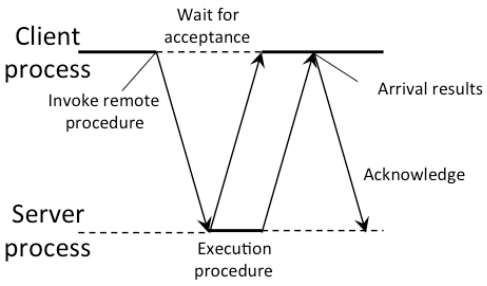

Figure 8. Deferred synchronous RPC [25]

calling environment is suspended, the parameters are passed across the network to the environment where the procedure is to be executed, and the desired procedure is executed there (Figure 7). The idea of RPC was older, but Birrel and Nelson were one of the first who implemented it [26]. This concept is further elaborated with the standards CORBA (Common Object Request Broker Architecture [27]) and DCOM (Distributed Component Object Model [28]). Also, the OPC Foundation based its first interoperability standard for industrial automation on DCOM. This first family of specifications is referred to as 'the classic OPC specifications' [29].

The ignorance on the part of the client about the fact that the server is located in a remote address space, was considered advantageous [25]. The client made use of a (local) library, which is dedicated for making a connection with a remote library, which was performing some tasks on the server side. Both libraries collaborate on a rather complicated mechanism to convert the client call to a message, and unpacking this message at the server side and convert it to a (local) call at the server side. All the details of the message passing are hidden away in the two libraries. Because of the message passing, this is message coupling, but for the user it looks like data or stamp coupling. Since the user cannot know whether there is a message coupling behind the data or stamp coupling, using or not using the concerned module cannot be a well considered choice or decision.

We evaluate that on top of the problems explained in the previous subsection about libraries and packages (subsection V-C) this concept, shown in Figure 7, is a violation of the separation of states theorem. Remember that a local module call is based on and thus dependent on the local address space. Hiding this dependency for the user also hinders the potential control over this dependency or assumption. For a local call, a fast reaction of the called module is assumed. For a remote call, the extra transfer time is not always negligible. Consequently, the suspension of the client during the call might be unfeasibly long. Also, when a communication failure occurs, the reply will not come at all, and the client will wait forever. In addition, the 'assumption' of the client that the call is local, does not discourages the user to pass variables by reference. While passing variables by reference assumes a local address space, this concept is not ideal in a remote call. When crossing the borders of a memory address space, each side of the coupling has to keep its own state. In other words, a reference to an item in an address space will become meaningless if the reference address is moved to another address space (and similar to content coupling). This would be an occurrence of a violation of the separation of concerns principle. In addition, because the value behind the reference is not copied in the respectively address spaces, we have a violation of the separation of states principle.

E. Synchronous versus asynchronous message passing

The concept of Figure 7, i.e., the client waits until the server replies before carrying on with its task, is called synchronous RPC. The action of communication on the client side can be summarized in one single line of programming code, there is a synchronization point between sender and receiver on message transfer. To minimize the 'wait for result' time, the concept of asynchronous RPC is introduced, where the client is not waiting for the reply, but only on an 'acceptance request' message. In combination with a similar call coming from the server (a so-called 'callback'), the client can receive the return results from the remote procedure in a comparable time frame as with synchronous RPC, but then without being blocked all the time (Figure 8). In comparison with synchronous communication, asynchronous communicates requires buffering to enable the program proceeding at the client side between request and reply. Before indicating this as an disadvantage, one should be aware that this buffering is exactly what the separation of states principle calls for. However, this principle is still not totally met, because the program at the client side can still hang when the 'acceptance request' message does not come, e.g., because of a network failure.

In the classic OPC specifications, both synchronous and asynchronous reading/writing functionality is available. However, experts indicated as a heuristic rule that asyn-

chronous communication is preferable. Indeed, the authors of the new family of interoperability standards for industrial automation, i.e., the OPC Unified Architecture (OPC UA), have abandoned the synchronous communication concept [30]. Instead, the OPC UA based communication is asynchronous by definition [31]. In terms of Normalized Systems, asynchronous communication reaches further towards complying the separation of states principle. In DCOM, there was an attempt to handle the risk that the client hangs when the 'acceptance request' message does not come by introducing a time-out mechanism. However, experts of the OPC Foundation reflected, based on worldwide surveys, that practitioners still call this an issue (note that classic OPC is based on DCOM). Lange et al. state that the time-out of DCOM in case of communication failures is too long, and not configurable [32].

We evaluate further that RPC, and DCOM, do not exhibit version transparency. Any change to a server requires all (remote) clients to have corresponding updates. When the size of a (distributed) system grows, this becomes infeasible because of the occurring combinatorial effects.

F. Service based communication

Services are modular constructs for aggregating software. Internally, they consist of modules, and they have one or more modular interfaces, that is accessible to the outside world. The basic idea is that some client application can call the services as provided by a server application. This principle is very similar to what was aimed at with remote procedure calls, except that the message coupling part is not hidden for the user. Services were first proposed in terms of web services, as they adhere to a collection of standards that will allow them to be discovered and accessed over the Internet. However, the term service has become more broadly interpreted later on. A service refers to technology-independent modules, implementable in different ways, including web services.

Web services are described by means of the Web Service Definition Language (WSDL) which is a formal language, comparable with the interface definition languages used to support RPC-based communication. A core element of a web service is the specification of how communication takes place. To this end, the Simple Object Access Protocol (SOAP) is used, which is essentially a framework in which much of the communication between two processes can be standardized [25]. Strange as it may seem, a SOAP envelope does not contain the address of the recipient. Instead, SOAP specifies bindings to underlying transfer protocols. In practice, most SOAP messages are sent over HyperText Transfer Protocol (HTTP). All communication between a client and server takes place through messages. HTTP recognizes only request and response messages. For our evaluation, a key field in the request line of the request message and status line of the response message is the version field. In other words, HTTP exhibits version transparency. Client and server can negotiate with the 'upgrade' message header on which version they will proceed. SOAP is designed with the assumption that client and server know very little of each other. Therefore, SOAP messages are largely based on the Extensible Markup Language (XML), which is on top of a markup language also a meta-markup language. In other words, in an XML description the syntax as used for a message is part of that message. This makes XML more flexible than the fixed markup language HyperText Markup Language (HTML), which is the most widely-used markup language in the Web.

Web services can be considered as a successor to RPC, like OPC UA (based on services) is a platform- and technology independent 'alternative' for classic OPC (based on DCOM). We doubt to use the word 'alternative' here, because classic OPC and OPC UA are complementary. Indeed, services can internally consist of classes or components, including DCOM based constructs. Web services separate software components from each other. They enable self-describing, modular applications to be published, located, and invoked across the web. Being a standardized interface, OPC UA enables interoperability between automation systems of different vendors. The industrial working groups of the OPC Foundation introduced *a mechanism to bring interoperability on an abstract level, without leaving the practical implementability*. To achieve this ambitious goal, they emphasized the importance of a communication context, and made a connection management concept between clients and servers mandatory. Probably OPC UA is also implementable for interoperability in other sectors than industrial automation [31].

The concept of asynchronous web-based messaging allows clients to proceed functioning, even if the server does not respond. From a technical point of view, a client can just carry on based on its own state. From a functional point of view, OPC UA incorporated mechanisms of notification and keep-alive messages to enable handling communication or remote system failures. This complies with the separation of states principle. The version tag in the HTTP messages enables compliance with the version transparency theorems.

VI. Summary of evaluations and guidelines

The core recommendation of this paper is making hidden dependencies explicit in the module's interface. In other words, safe black box (re)use requires that a developer is able to anticipate which conditions are necessary for (re)use. A self-explaining interface is a good start, but typically dependencies like packages, libraries, global variables, implicitly used communication technologies, references to a local address space, are not included in the interface. We conclude that it should, and phrase the following rule.

In order to design safe black box (re)useable software components, every (re)use of a library, package, global variable or implicitly use of a communication technology in a module, should include a declaration, reference, path or link to the identification of the dependency, accompanied with the used version.

We make the reflection that there is a similarity between global variables, which are not declared with the 'external' keyword and other dependencies, which are not declared in the module's interface. It can be interpreted that these dependencies can cause common coupling. Hiding these dependencies makes it impossible to evaluate them and let the user decide whether these dependencies can or cannot be made available in the environment in which the user is considering them to (re)use. Note that declarations to make these dependencies visible should include the versions of the external constructs, to prevent combinatorial effects in case of updates, and to enable the co-existence of different versions of the same core constructs in a library or external technology.

In addition to our rather general rule, we define some explicit guidelines:

1) Explicitation of global variables: Global variables should be treated as local variables of the main program, and passed to called modules by reference or via the in/out variables in an IEC 61131-3 environment. These variables could be passed further in cascade to submodules called by modules, where they are locally always treated as in/out variables.

Application example: Consider an IEC 61131-3 Function Block which is controlling a motor. This Function Block (FB) is calling other FBs on submodular lever, where the core functionality is a state machine of the motor. In addition, there are supporting FBs on submodular level, which provide functionality to manage manual/automatic mode, alarming, interlocking, hardware connection, and simulation. The FB on modular level (dispatching task) receives a data struct, which contains all the states, commands, and hardware IOs of both core and supporting functionality. This data struct is a global variable. The dispatching FB calls FBs on submodular level and passes the data struct to each of the supporting FBs as an in/out variable. This design has a modular structure with a high granularity. Since the functionality of the FBs on submodular level is limited and generic, the reuse potential is high.

2) Pass by reference should strictly adhere to one single address space: In/out variables, passed by reference, loose their meaning in another address space. Therefore, the pass by reference concept should be limited to the same environment or address space where the referred variable is defined. In case it is desired to cross the borders of the address space, a copy of the concerned variable or a pass by value is required.

Application example: Consider the same data structure which contains all the data about a motor. This data structure is defined as a global construct, and is passed to the dispatching FB by reference. This reference is passed further on submodular level to the supporting FBs. Now, outside the PLC, a low level HMI (Human Machine Interface) application is used to control the motor on submodular level on a Windows PC. This Windows PC cannot use the reference, which is only meaningful in the PLC. Instead, the entire data structure is copied via an OPC interface (message coupling) to the HMI application.

3) Explicitation of external modules: Couplings to external modules can be (re)used, library modules included, but they should be declared in a similar way like the 'external' keyword for global variables, including the path of the communication context. In other words, library management should be done on the level of the module, not on the level of the programming environment. In addition, the versions of the called modules should be declared.

Application example: Our data structure is defined as a global IEC 61131-3 configuration. In the main program, this is not visible, unless this data structure is declared as an external defined data structure in the main program (POU). As such, the data structure can be treated as local for the main program.

4) Abstraction of external technologies: It is allowed to hide information about an external technology, but an abstraction of the core functionality should be declared, including the fact that this functionality is abstract, and relying on a remote technology. The entity which is managing the connection with this abstract remote technology should exhibit state keeping, and notify autonomously unexpected behavior of the remote technology.

Application example: Suppose the motor is controller with a frequency drive. We do not have control over potential firmware updates of this frequency drive. It is also possible that at some moment in time the frequency drive will be replaced by another type or brand. Therefore, we include in data struct fields which are representing the core functionality like setpoint, ramp, speed, current, etc. A connection entity is responsible to convert the representation or data type of these fields. For every version another connection entity has to be written. A connection element selects the appropriate version based on a version ID.

VII. Conclusion

The reasons why properties like evolvability, (re)usability, and safe black box design are difficult to achieve, have most likely something to do with a lack of making the existing knowledge and experience-based guidelines on sound modular design explicit. Undoubtedly, the theorems of Normalized Systems contribute on this issue by formulating unambiguous design rules at the elementary level of software primitives. On a higher implementation level, it is expected that

not all implementation questions like those related to, e.g., a dependency-hell, are easy to answer. Experienced engineers will find that these are violations of the theorems 'separation of concerns' and 'separation of states'. However, for less experienced engineers, more practical oriented examples or manifestations of violations and how to avoid them, seem useful as well. We aim that — on top of these fundamental principles — some derived rules can make these violations easier to catch, also for less experienced engineers.

In this paper, we introduced the derived rule that any dependency should be visible in the module's interface, accompanied by its state and version. The way how this information is included in the interface, should be done in a version transparent way, to prevent violations of the 2nd and 3rd principle of Normalized Systems.

We made a study of a set of different kind of couplings on an abstract way, and evaluated these types of couplings against the Normalized Systems theorems. In addition, implications arise when modules are placed in an address space, based on a paradigm or construct in a concrete programming environment. Special attention is needed when a module, placed in the local address space, is coupled with another module, which is placed in a remote address space. After evaluating these implications, we derived four guidelines towards better controlling dependencies.

We designed the derived rules with the potential to become generic, independent of the application domain. As a first start, we exemplified the rules and analyses in a PLC (IEC 61131-3 based) environment. In future work, our aim is to investigate to which extent these rules can be implemented in other technologies and programming environments as well.

ACKNOWLEDGMENT

P.D.B. is supported by a Research Grant of the Agency for Innovation by Science and Technology in Flanders (IWT).

REFERENCES

[1] D. van der Linden, H. Mannaert, and P. De Bruyn, "Towards the explicitation of hidden dependencies in the module interface," in *ICONS 2012, 7th International Conference on Systems*, 2012.

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

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

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

[5] H. Mannaert, J. Verelst, and K. Ven, "Exploring the concept of systems theoretic stability as a starting point for a unified theory on software engineering," in *ICSEA 2008, 3th International Conference on Software Engineering Advances*, 2008.

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

[7] 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.

[8] ——, "Towards evolvable software architectures based on systems theoretic stability," *Software: Practice and Experience*, vol. 42, no. 1, pp. 89–116, 2012.

[9] 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.

[10] G. Myers, *Reliable Software through Composite Design*. Van Nostrand Reinhold Company, 1975.

[11] Wikipedia, "Coupling (computer programming)," *Wikipedia*, last accessed June 2013. [Online]. Available: http://en.wikipedia.org/wiki/Coupling_(computer_programming)

[12] D. Van Nuffel, H. Mannaert, C. De Backer, and J. Verelst, "Towards a deterministic business process modelling method based on normalized theory," *International journal on advances in software*, vol. 3, no. 1 and 2, pp. 54 – 69, 2010.

[13] E. Dijkstra, "Go to statement considered harmful," *Communications of the ACM 11(3)*, pp. 147 – 148, 1968.

[14] S.-M. Huang, C.-F. Tsai, and P.-C. Huang, "Component-based software version management based on a component-interface dependency matrix," *Journal of Systems and Software*, vol. 82, no. 3, pp. 382 – 399, 2009.

[15] T. D. Vu, "Goto elimination in program algebra," *Science of Computer Programming*, vol. 73, no. 2 - 3, pp. 95 – 128, 2008.

[16] E. W. Dijkstra, "Structured programming," O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. London, UK, UK: Academic Press Ltd., 1972, ch. Chapter I: Notes on structured programming, pp. 1–82.

[17] D. J. W. Maurice V. Wilkes and S. Gill, *The preparation of programs for an electronic digital computer*. Addison-Wesley Press, 1951.

[18] *Programming with STEP7*, Siemens, 05 2010.

[19] "ALGOL 60," last accessed June 2013. [Online]. Available: http://en.wikipedia.org/wiki/ALGOL_60

[20] D. Nykamp, "Function machine parameters," last accessed June 2013. [Online]. Available: http://mathinsight.org/function_machine_parameters

[21] I. Kuhl and A. Fay, "A middleware for software evolution of automation software," *IEEE Conference on Emerging Technologies and Factory Automation*, 2011.

[22] D. van der Linden, G. Neugschwandtner, and H. Mannaert, "Industrial automation software: Using the web as a design guide," in *ICIW 2012, 7^{th} International Conference on Internet and Web Applications and Services*.

[23] M. de Sousa, "Proposed corrections to the IEC 61131-3 standard," *Computer Standards & Interfaces*, pp. 312–320, 2010.

[24] K. Thramboulidis and G. Frey, "An MDD process for IEC 61131-based industrial automation systems," in *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, sept. 2011, pp. 1 –8.

[25] A. Tanenbaum and M. Van Steen, *Distributed Systems: principles and paradigms*. Pearson Prentice Hall, 2007.

[26] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39 – 59, 1984.

[27] "Corba," last accessed June 2013. [Online]. Available: http://www.omg.org/spec/CORBA/

[28] G. Eddon and H. Eddon, *Inside Distributed COM*. Microsoft Press, 1998.

[29] *OPC DA Specification*, OPC Foundation Std. Version 2.05a, 2002.

[30] "OPC Unified Architecture Specifications," last accessed June 2013. [Online]. Available: http://www.opcfoundation.org

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

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