On the Generalization of Normalized Systems Concepts to the Analysis and Design of Modules in Systems and Enterprise Engineering

Peter De Bruyn and Herwig Mannaert
Department of Management Information Systems
Normalized Systems Institute (NSI)
University of Antwerp
Antwerp, Belgium
{*peter.debruyn, herwig.mannaert*}@*ua.ac.be*

***Abstract*—Current organizations need to be able to cope with challenges such as increasing change and increasing complexity in many or all of their aspects. Modularity has frequently been suggested as a powerful means for reducing that complexity and enabling flexibility. However, the proper use of modularity to actually achieve those benefits cannot be considered trivial or straightforward. Normalized Systems (NS) theory has proven to introduce this evolvable and diagnosable modularity in software systems. This paper discusses the generalization of NS concepts to the analysis and design of modules in systems and enterprise engineering as evolvability and diagnosability are deemed to be appealing for most modular structures. In order to do so, this paper highlights the importance of distinguishing blackbox and whitebox views on systems and the fact that a true blackbox requires fully and exhaustively defined interfaces. We further discuss the functional/constructional transformation and elaborate on how NS theory uses the concepts of modularity, stability and entropy to optimize certain properties of that transformation. We argue how some aspects of organizational systems can be analyzed based on the same reasoning, suggesting some viable approaches for Enterprise Engineering. By means of a tentative reflection, we provide a discussion regarding how the concepts of stability and entropy might be interpreted as different manifestations of coupling within modular structures.**

***Keywords*-Normalized Systems; Systems Engineering; Enterprise Engineering; Modularity; Stability; Entropy.**

I. Introduction

Current organizations need to be able to cope with increasing change and increasing complexity in many or all of their aspects. Not only organizations themselves need to deal with this '*changing complexity*' in terms of their organizational structures, business processes, etcetera. Additionally, also the products or services they deliver, and even the software applications supporting these products and business processes, are equally exposed to this changing complexity. In many engineering disciplines, *modularity* has previously been suggested as a powerful means for reducing that complexity by decomposing a system into several subsystems [2], [3]. Moreover, modifications at the level of those subsystems instead of the system as a whole are said to facilitate the overall evolvability of the system. Hence, modularity can be claimed to have properties for tackling both change and complexity in systems.

However, the proper use of modularity to actually achieve those benefits cannot be considered trivial or straightforward. Typical issues involved include the identification and delineation of the modular building blocks (i.e., subsystems), the communication between those modular building blocks (i.e., the interfaces), the assurance of providing compatibility of new versions of a building block with the overall system, etcetera.

In this regard, *Normalized Systems (NS) theory* has recently proven to introduce this diagnosable and evolvable modularity, primarily at the level of software systems [4], [5]. Considering the transformation of basic functional requirements into software primitives (such as data structures, functions, methods, etcetera), which are considered as the basic modular building blocks of software systems, the theory proposes a set of formally proven theorems to design and analyze software architectures. Besides using the concept of modularity, the theory also heavily relies on other traditional engineering concepts such as stability (based on systems theory) [4] and entropy (based on thermodynamics) [6] to optimize those modular structures according to certain criteria. As this approach has proven its value in the past to obtain more maintainable and diagnosable software architectures, it seems appealing to investigate the extent to which we can apply the same approach to other modular systems for which these properties seem beneficial as well.

Indeed, we claim that many other systems could also be regarded as modular structures. Both functional (i.e., requirements) and constructional (i.e., primitives) perspectives can frequently be discerned, modules can be identified and thus the analysis of the functional/constructional transformation seems relevant. For instance, considering organizational systems, Van Nuffel has recently shown the feasibility of applying modularity and NS theory concepts at the business process level [7], [8] while Huysmans did so at the level of enterprise architectures [9]. However, the extension of NS theory to these domains has not been fully formalized yet in several aspects. Consequently, as NS theory proved

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

to be successful in introducing evolvable and diagnosable modularity in software systems, and as it is clearly desirable to extend such properties to other systems, this paper focuses on applying NS theory concepts to the identification and analysis of modular structures in systems engineering (including Enterprise Engineering). This way, the paper can be regarded as a first step towards the formal generalization of NS theory concepts to modularity and system engineering in general.

More specifically, we will focus in the present paper on the illustration and discussion of the following aspects:

1) From an engineering perspective, arguably, many systems can be considered as *modular structures*, including traditional engineering systems, software systems, and organizational systems;
2) In order to profoundly study (and in a second phase, optimize) the structure of such modular systems, it is necessary to formulate *complete and exhaustive interfaces* for each of their constituting modular building blocks as this gives a complete overview of the coupling between them (and, possibly, external systems). This is considered to be an essential part of each systems engineering process;
3) As proposed by Normalized Systems theory, traditional engineering concepts such as *stability* (based on systems theory) and *entropy* (based on thermodynamics) might offer interesting viewpoints for the analysis and optimization of modular systems, each from their own perspective (i.e., evolvability and diagnosability respectively);
4) One way of interpreting both the occurrence of instability and entropy in relation to modular structures, is to consider *coupling* between modular components as their common origin and ground. This further adds to our discussion regarding the importance of fully defined and complete interfaces, as argued under bullet point 2.

This paper mainly further elaborates the reasoning proposed in [1] regarding the need for complete and unambiguous module interfaces (i.e., bullet points 1 and 2) by explicitly relating them to the concepts of coupling, stability and entropy (i.e., bullet points 3 and 4). Also, additional examples (i.e., cases) regarding the consequences of applying our reasoning to organizational systems, will be provided.

The remainder of this paper is structured as follows. Section II discusses the essence of NS theory and how it leverages the concepts of stability and entropy to obtain evolvable and diagnosable modular structures. Next, in Section III, we present some extant literature on modularity (without claiming to be exhaustive), emphasizing the work of Baldwin and Clark. Here, some arguments will also be offered to consider it reasonable to analyze both software and organizational systems from a modularity point of view. Afterwards, we differentiate between blackbox (functional) and whitebox (constructional) views on systems, and offer a more unambiguous definition of modularity by arguing for the need of complete and exhaustively defined interfaces in Section IV. Some useful functional/constructional transformation properties (including stability and entropy) will be discussed in Section V. Emphasizing the usefulness of our approach for Enterprise Engineering, the application of NS stability and entropy reasoning to business processes will be illustrated in Section VI, as well as some additional interface dimensions, which could show up when considering organizational modules (ideally exhibiting a complete interface). In Section VII, some additional examples (i.e., cases) regarding the consequences of applying our reasoning to organizational systems, will be provided. We end this paper by reflecting on the relatedness of the concepts of stability and entropy in terms of modular coupling (Section VIII) and some conclusions (Section IX).

II. Normalized Systems theory

Normalized Systems theory (NS) is a theory about the deterministic creation of evolvable modular structures based on a limited set of proven and unambiguous design theorems, primarily aimed at the design of evolvable software architectures. In order to do so, the theory states that the implementation of functional requirements into software constructs can be regarded as a *transformation* of a set of requirements $\mathcal{R}$ into a set of software primitives $\mathcal{S}$ [10], [4], [5]:

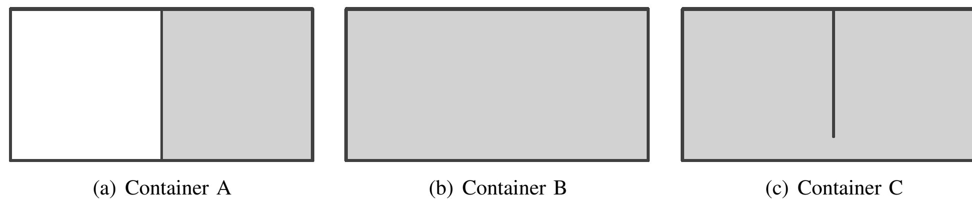
$$\{\mathcal{S}\} = \mathcal{I}\{\mathcal{R}\}$$

Next, the theory argues that the resulting set of primitives can be considered to be a *modular structure* and that ideally (1) the considered design-time transformation should exhibit stability (i.e., evolvability), and (2) the run-time instantiation of implemented primitives should exhibit isentropicity (i.e., diagnosability). For this purpose, a set of theorems is derived based on insights from traditional engineering sciences such as systems theory and thermodynamics. In this section, we will briefly highlight both approaches. First, we will discuss the essence of NS in its initial form, i.e., starting from the stability point of view from systems theory. Next, the recent association and indications towards conformance with entropy concepts from thermodynamics will be highlighted. A preliminary discussion of some real-life NS software implementations can be found in [5].

A. Normalized Systems and Stability

Normalized Systems theory initially originated from the well-known maintenance problems in software applications and the phenomenon that software programs tend to become ever more complex and badly structured as they are changed over time, becoming more and more difficult to adapt and

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org



(a) Container A (b) Container B (c) Container C

Figure 1. Container example for illustrating the concept of entropy. The grey parts represent those spaces filled with gas, the white parts represent those spaces, which are empty.

hence, less evolvable [11]. In order to obtain evolvable modularity, NS theory states that the functional/constructional transformation should exhibit *systems theoretic stability*, meaning that a bounded input function (i.e., bounded set of requirement changes) should result in bounded output values (i.e., a bounded impact or effort) even if an unlimited systems evolution with time $t \rightarrow \infty$ is considered. From this perspective, Mannaert et al. [4] have formally proven that this implies that the modular structure should strictly adhere to the following *principles*:

- *Separation of Concerns (SoC)*, enforcing each concern (here: change driver) to be separated;
- *Data Version Transparency (DvT)*, enforcing communication between data in a version transparant way;
- *Action Version Transparency (AvT)*, requiring that action components can be updated without impacting calling components;
- *Separation of States (SoS)*, enforcing each action of a workflow to be separated from other actions in time by keeping state after every action.

As the systematic application of these principles results in very fine-grained modular structures, NS theory proposes to build information systems based on the aggregation of instantiations of five higher-level software patterns or *elements*, i.e., action elements, data elements, workflow elements, trigger elements and connector elements [10], [4], [5]. Typical cross-cutting concerns (such as remote access, persistence, access control, etcetera) are included in these elements in such a way that it is consistent with the above-mentioned theorems.

A more formal discussion of the stability principles and reasoning as well as some initial case study findings can be found in [4] and [5] respectively.

B. Normalized Systems and Entropy

Recently, efforts were made to explain the above-mentioned findings in terms of *entropy as defined in thermodynamics* [6]. Entropy is a well-known and much debated engineering concept, originating from thermodynamics and referring to its Second Law. As we pointed out in [12], some common interpretations associated with entropy include (1) complexity, perceived chaos or disorder [13], (2) uncertainty or lack of information [14] and (3) the tendency of constituent particles in a system to dissipate or spread out [15]. In [6], it was proposed to primarily employ the statistical thermodynamics perspective on entropy for the extension of NS theory. As such, the definition of Boltzmann [16] was adopted, considering entropy as the number of microstates (i.e., the whole of microscopic properties of a system) consistent with a certain macrostate (i.e., the whole of externally observable and measurable properties of a system).

Consider for example Figure 1, symbolizing a gas container having a boundary in the middle, which is dividing the container into two compartments. The boundary completely isolates both parts of the container as a result of which the gas is solely present in the right side of the container, leaving the left part empty (see Panel (a)). While the macrostate of the container (e.g., its temperature or pressure) is brought about by one particular arrangement or configuration of the gas molecules (i.e., its microstate: the union of the position, velocity, and energy of all gas molecules in the container), many different configurations of molecules (microstates) might result in this same macrostate (hence illustrating the amount of entropy). Now imagine that the shaft between the two components is removed and both components become one single space: the gas (and the energy of its molecules) will expand, dissipate and spread out into the full space, interacting with the second component of the container (see Panel (b)). This interaction (and the removal of the fragmentation, separation and structure between both spaces) moreover increases the degree of entropy as now even a larger set of microstates (configurations of the molecules) can result in a single macrostate. The only way to avoid an increase of entropy throughout time is by introducing structure or effective boundaries between subsystems. Note that such boundaries need to be complete and encompassing, as in the case of partial fragmentation no entropy reduction is obtained (see Panel (c)).

Applying this reasoning to software applications, *microstates* could be defined as binary values representing the correct or erroneous execution of a construct of a programming language. A *macrostate* can then be seen in terms of loggings or database entries representing the correct or erroneous processing of the considered software system. From this perspective, Mannaert et al. [6] have proposed a second set of principles that should be strictly adhered to in order to achieve diagnosability in a modular structure. First,

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

the above described principles of *Separation of Concerns* (now in terms of information units) and *Separation of States* could equally be derived from this entropy reasoning as well. Next, a set of two additional principles were formulated:

- *Data instance Traceability (DiT)*, enforcing the actual version and the values of every instance of a data structure serving as an argument, to be exported (e.g., logged) to an observable macrostate;
- *Action instance Traceability (AiT)*, enforcing the actual version of every instance of a processing function and the thread it is embedded in, to be exported (e.g., logged) to an observable macrostate.

III. Related Work on Modularity

The use of the concept of modularity has been noticed to be employed in several scientific domains such as computer science, management, engineering, manufacturing, etcetera [2], [3]. While no single generally accepted definition is known, the concept is most commonly associated with the process of subdividing a system into several subsystems [19], [20]. This decomposition of complex systems is said to result in a certain degree of complexity reduction [21] and facilitate change by allowing modifications at the level of a single subsystem instead of having to adapt the whole system at once [22], [2], [3].

As such, Baldwin and Clark defined modularity as follows: "*a unit whose structural elements are powerfully connected among themselves and relatively weakly connected to elements in other units*" [3, p. 63]. They conceive each system or artifact as being the result of specifying values for a set of design parameters, such as the height and the vessel diameter in case of a tea mug. The task of the designer is then to choose the design parameter values in such a way, that the 'market value' of the system as a whole becomes maximized. Some of the design parameters might be dependent on one another, as for example the value of the vessel diameter should be attuned to the value of the diameter of a mug. This reasoning is visualized in Figure 2 as well. Here, the use of Design Structure Matrices (DSM) is proposed. Such matrices (as originally elaborated by Steward [17] and Eppinger et al. [18]) typically depict the design parameters in both the rows and columns of the matrix. Each 'x' represents a dependency (hence, coupling) between two design parameters. Additionally, the direction of the dependency is indicated: for instance, in the situation as depicted in Figure 2, the matrix implies that the choice of a particular value for design parameter B (e.g., mug diameter) determines the set of possible choices for the value of design parameter A (e.g., vessel diameter). Obviously, a myriad of types of Design Structures Matrices can appear, according the system under consideration.

After drafting a Design Structure Matrix for the system to be engineered (i.e., providing a detailed overview of the dependencies among the different relevant design parameters), modularization is conceived by Baldwin and Clark as the process in which groups of design parameters — highly interrelated internally, but loosely coupled externally — are to be identified as modules and can be designed rather independently from each other, such as for instance the drive system, main board and LCD screen in case of a simplified computer hardware design. In Figure 2, the different modules are indicated by the light and dark grey zones, respectively. As argued by Baldwin and Clark, the modules should thus be ideally fully decoupled: this would mean that, as is the case in Figure 2, there are only 'x's placed within the light and dark grey zones and no 'x's should be found in the white areas representing dependencies between both modules. Nevertheless, as such a situation is mostly only a theoretical ideal, in most realistic settings, dependencies between the distinct modules do occur. In such situations, a set of architectural or design rules (i.e., externally visible information) is typically used to secure the compatibility between the subsystems in order to be assembled into one working system later on, while the other design parameters are only visible for a module itself. Finally, they conclude that this modularity property allows for multiple (parallel) experiments for each module separately, resulting in a higher 'option value' of the system in its totality. Instead of just accepting or declining one system as a whole, a 'portfolio of options' can be considered, as designers can compose a system by purposefully selecting among a set of alternative modules. Systems evolution is then believed to be characterized by the following six modular operators [3]:

- *Splitting* a design (and its tasks) into modules;
- *Substituting* one module design for another;
- *Augmenting*, i.e., adding a new (extra) module to the system;
- *Excluding* a module from the system;
- *Inverting*, i.e., isolating common functionality in a new module, thus creating new design rules;
- *Porting* a module to another system.

Typically, besides traditional physical products, many other types of systems are claimed to be able to be regarded as modular structures as well. First, all different programming and software paradigms can be considered as using modularity as a main concept to build software applications [10]. Whether they are using classes, objects, structures, functions, procedures, etcetera, they all are basically allowing a programmer to compose a software system by aggregating a set of instances from a collection of primitives (available in the concerning programming paradigm) in a modular way. Furthermore, while Baldwin and Clark primarily illustrate their discussion by means of several evolutions in the computer industry, they also explicitly refer to the impact of product modularity on the (modular) organization of workgroups both within one or multiple organizations,

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

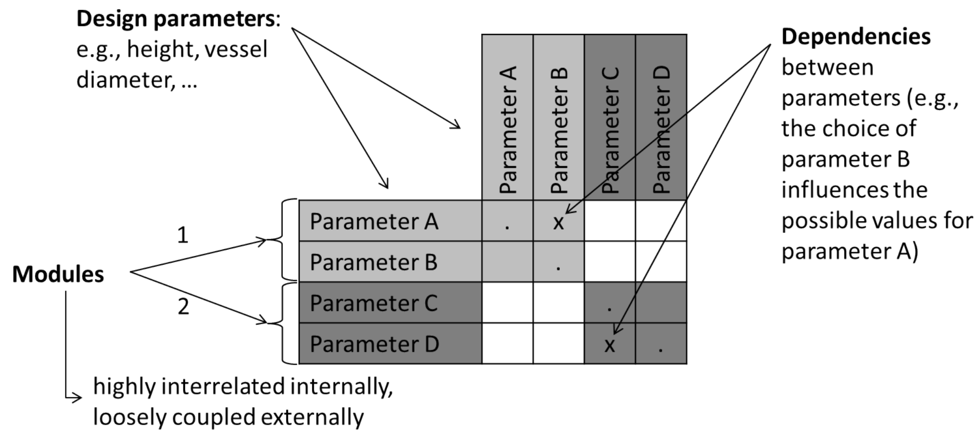


Figure 2. Modularity reasoning as proposed by Baldwin and Clark [3], based on Design Structure Matrices (DSM) from Steward [17] and Eppinger et al. [18].

and even whole industry clusters [3]. Also, Campagnolo and Camuffo [20] investigated the use of modularity concepts within management science and identified 125 studies in which modularity concepts arose as a design principle of complex organizational systems, suggesting that principles based on the concept of modularity offer powerful means to be applied at the organizational level.

Within the field of Enterprise Engineering, trying to give prescriptive guidelines on how to design organizations according to certain (desirable) characteristics, modularity equally proved to be a powerful concept. For instance, Op't Land used modularity related criteria (including coupling and cohesion) to analyze and predict the merging and splitting of organizations [23]. Van Nuffel proposed a framework to deterministically identify and delimit business processes based on a modular and NS theory viewpoint [7], [8], and Huysmans demonstrated the usefulness of modularity with regard to the study of (the evolvability) of enterprise architectures [9].

IV. Towards a Complete and Unambiguous Definition of Modules

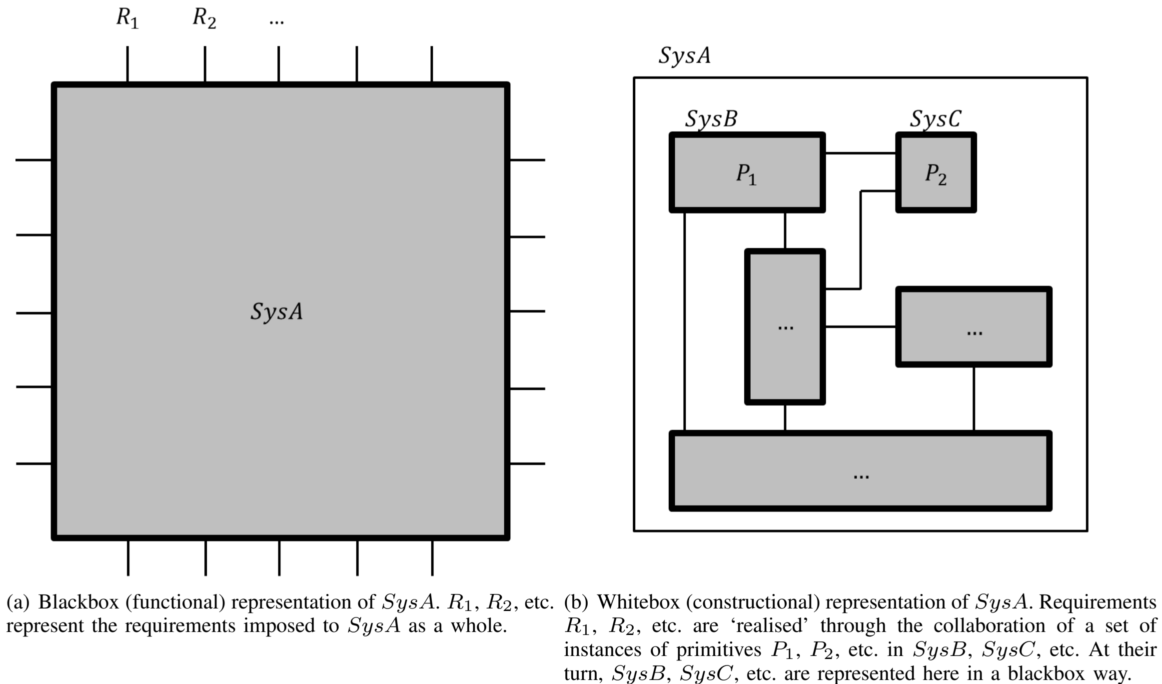
While we are obviously grateful for the valuable contributions of the above mentioned authors, we will argue in this section that the definition of modularity, as for example coined by Baldwin and Clark [3], already describes an ideal form of modularity (e.g., loosely coupled and independent). As such, before starting our generalization efforts of NS theory to modularity issues in general, we need to clarify a few elements regarding our conceptualization of modularity. First, we will discuss the need to distinguish both functional and constructional perspectives of systems. Next, we will propose to introduce the formulation of an exhaustive modular interface as an intermediate stage, being a necessary and sufficient condition in order to claim 'modularity'. The resulting modules can then be optimized later on, based on particular criteria, which will be our focus of Section V.

A. Blackbox (Functional) versus Whitebox (Constructional) Perspectives on Modularity

When considering systems in general — software systems, organizational systems, etcetera — both a functional and constructional perspective should be taken into account [24]. The *functional perspective* focuses on describing what a particular system or unit does or what its function is [25]. While describing the external behavior of the system, this perspective defines input variables (what does the system need in order to perform its functionality?), transfer functions (what does the system do with its input?) and output variables (what does the system deliver after performing its functionality?). As such, a set of general requirements, applicable for the system as a whole, are listed. The *constructional perspective* on the other hand, concentrates on the composition and structure of the system (i.e., which subsystems are part of the system?) and the relation of each of those subsystems (i.e., how do they work together to perform the general function and adhere to the predefined requirements?) [26].

Equivalently, one could regard the functional system view as a blackbox representation, and the constructional system view as a whitebox representation. By *blackbox view* we mean that only the input and output of a system is revealed by means of an interface, describing the way how the system interacts with its environment. As such, the user of the system does not need to know any details about the content or the inner way of working of the system. The way in which the module performs its tasks is thus easily allowed to change and can evolve independently without affecting the user of the system, as long as the final interface of the system remains unchanged. The complexity of the inner working can also be said to be hided (i.e., information

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org



(a) Blackbox (functional) representation of $SysA$. R_1, R_2, etc. represent the requirements imposed to $SysA$ as a whole.

(b) Whitebox (constructional) representation of $SysA$. Requirements R_1, R_2, etc. are 'realised' through the collaboration of a set of instances of primitives P_1, P_2, etc. in $SysB$, $SysC$, etc. At their turn, $SysB$, $SysC$, etc. are represented here in a blackbox way.

Figure 3. Blackbox (functional) and whitebox (constructional) representations of system $SysA$.

hiding), resulting in some degree of complexity reduction. The *whitebox view* does reveal the inner way of working of a system: it depicts the different parts of which the system consists in terms of primitives, and the way these parts work together in order to achieve the set of requirements as listed in the blackbox view. However, each of these parts or subsystems is a 'system' on its own and can thus again be regarded in both a functional (blackbox) and constructional (whitebox) way.

The above reasoning is also depicted in Figure 3: both panels represent the same system $SysA$, but from a conceptually different viewpoint. Panel (a), depicting the functional (blackbox) view, lists the requirements (boundary conditions) R_1, R_2, $\ldots$ imposed to the system. These are proposed as 'surrounding' the system in the sense that they do not say anything about how the system performs its tasks, but rather discuss what it should perform by means of an interface in terms of inputs and outputs. Panel (b) depicts the constructional (whitebox) view of the same system: the way of working of an aggregation of instantiations of primitives P_1, P_2, $\ldots$ (building blocks), collaborating to achieve the behavior described in Panel (a). Each of the primitives in Panel (b) is again depicted in a blackbox way and could, at their turn, each also be analyzed in a constructional (whitebox) way.

B. Avoiding Hidden Coupling by Strictly Defining Modular Interfaces

Before analyzing and optimizing the transformation between both perspectives, the designer should be fully confident that the available primitives can really be considered as 'fully fledged, blackbox modules'. By this, we mean that the user of a particular module should be able to implement it (i.e., in design-time), exclusively relying on the available interface, thus without having any knowledge about the inner way of working of the concerned module. Stated otherwise, while other authors previously already elaborated on the importance of interfaces for allowing a workable modular design (see e.g., [27]), we primarily stress the importance that the interface of a module should describe any possible dependency regarding the module, needed to perform its functionality. As long as the interface or boundary is not fully articulated, undocumented interaction with other systems can and will occur. In such situation, the study of the functional/constructional transformation or any optimization effort may become worthless. To a certain extent, our reasoning is also somewhat parallel to the motivation of Dijkstra in his argument to abolish 'goto instructions' in programming languages [28]: an incomplete or underspecified interface makes it very difficult if not impossible for an engineer to mentally mimic the actual 'way of working' of the modules (i.e., at run-time) as no clear overview is available on how they affect one another. The importance of a full interface can finally be illustrated by Figure 1(c): only in case the

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

boundary (i.e., interface) is complete, a subsystem (here: gas container compartment) can be properly studied in an isolated way.

Consequently, every interaction of a system with its environment should be properly and exhaustively defined in the interface of a module. While this may seem rather straightforward at first sight, real-life interfaces are rarely described in such a way. Indeed, typical non-functional aspects such as technological frameworks, infrastructure, knowledge, etcetera are consequently also to be taken into account (see Section VI-B). Not formulating these 'tacit assumptions' in an explicit way results in hidden coupling: while the system is claimed to be a module, it actually still needs whitebox inspection in order to be implemented in reality, diminishing the pretended complexity reduction benefits.

Consider for instance a multiplexer for use in a typical processor, selecting and forwarding one out of several input signals. Here, one might conceptually think at a device having for example 8 input signals, 3 select lines and 1 output signal. While this is conceptually certainly correct, a real implementation on a real processor might for example (hypothetically) require 120μ by 90μ CMOS (i.e., material) to make the multiplexer physically operational on the processor, while this is not explicitly mentioned in its conceptual interface. As such, this 'resource dimension' should be made explicit in order to consider a multiplexer as a real blackbox in the sense that the module can be unambiguously and fully described by its interface. A person wanting to use a multiplexer in real-life in a blackbox way, should indeed be aware of this prerequisite prior to his ability of successfully implementing the artifact.

A more advanced example of hidden coupling includes the use of a 'method' in typical object-oriented programming languages, frequently suggested as a typical example of a 'module' in software. Indeed, in previous work, it was argued to consider the multidimensional variability when analyzing the evolvability of programming constructs (such as data structures and processing functions) and that in typical object-oriented programming environments these dimensions of variability increase even further as they make it possible to combine processing actions and data entities into one primitive (i.e., a single class) [4]. Hence, it was argued to start the analysis of object-oriented modular structures already at the level of methods instead of only considering a class as a possible 'module'. However, while it is usually said that such a method in object orientation has an interface, this interface is not necessarily completely, exhaustively and fully defined and thus such a method cannot automatically be considered as a 'real module' according to our conceptualization. Consider for example the constructor of the class in which the method has been defined. Typically, the constructor has to perform certain actions (e.g., making an instantiation (object) of the concerned class) before one can execute the concerned method. Also member variables of the class might introduce hidden coupling: first, they can be manipulated by other methods as well, outside control of the considered method. Second, they have to be created ('exist') before the module can perform its functionality. Finally, employing external libraries in case a method wants to be deemed a genuine module, would imply that ór the library should be incorporated into the module (each time) ór the external library should be explicitly mentioned in the interface. Analogously, a method having a clear interface regarding how to call the method and how the returned values should be captured, can still call (at its turn) another (third) method or rely on a certain external service or technology (e.g., a connection to the financial network of SWIFT, the use of an interface module to an external system, etcetera). As these extra methods, services or technologies are necessary to successfully complete the method, they should actually be included in such a complete and exhaustive interface in addition to the typical parameters and instructions needed to make the method call.

Hence, in our view, one has a genuine module as soon as one is able to define a complete interface, which clearly describes the boundaries and interactions of the subsystem and allows it to be used in a blackbox way. Modularization is then the process of meticulously identifying each dependency of a subsystem, transforming an ambiguously defined '*chunk*' of a system into a clearly defined *module* (of which the borders, dependencies, etcetera are precisely and ex-ante known). Compared to the definition of Baldwin and Clark cited previously, we thus do not require for a module to exhibit already high intramodular cohesion and low intermodular coupling at this stage. Modules having these characteristics are nevertheless obviously highly desirable. However, we are convinced that defining in a first phase such a complete interface, allows to 'encapsulate' the module in an appropriate way and avoid any sort of hidden coupling. Indeed, at least four out of the six mentioned modular operators in Section III require real blackbox (re)usable modules as a conditio sine qua non. More specifically, in order to use the operators Substituting, Augmenting, Excluding and Porting in their intended way, complete and exhaustively defined interfaces are a prerequisite. On the other hand, the modular operators Splitting and Inverting concern the definition of new modules and design rules. Hence, they are precisely focused on the process of defining new modular interfaces themselves, thus usually involving some form of whitebox inspection.

Finally, while defining modules with such a strict interface will not directly solve any interdependency, evolvability, ... issues, it will at least offer the possibility to profoundly study and optimize the 'quality' of the modules (e.g., with regard to coupling and cohesion) in a next stage.

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

V. Towards Generalizing Normalized Systems to the Function/Construction Transformation

In the same way as the implementation of software is considered as a transformation $\mathcal{I}$ of a set of functional requirements into a set of software primitives (constructs) in [4], the design or engineering of systems in general could be considered as a transformation $\mathcal{D}$ of a set of functional requirements R_j into a set of subsystems or primitives P_i:

$$\{P_i\} = \mathcal{D}\{R_j\}$$

This transformation $\mathcal{D}$ can then be studied and/or optimized in terms of various desirable system properties. In this section, we present a very preliminary discussion on the meaning of several important system properties in this respect.

First, it seems highly desirable to have a linear design transformation that can be *normalized*. This would imply that the transformation matrix becomes diagonal or in the Jordan form, leading to a one-to-one mapping of functional requirements to (a set of) constructional primitives. Such a normalized transformation is explored in [10], [4] for the implementation of elementary functional requirements into software primitives.

Moreover, this approach to systems design or engineering also seems to imply that we should avoid to perform functional decomposition over many hierarchical levels, before starting the composition or aggregation process [29]. Studying and/or optimizing the functional to constructional transformation is a very delicate activity that can only be performed on one or two levels at a time. Therefore, the approach seems to imply a preference for a bottom-up or meet-in-the-middle approach, trying to devise the required system (i.e., the set of functional requirements R_j) in terms instantiations of a set of predefined primitives (i.e., P_i), over a top-down approach.

Next, some other appealing transformation properties might include stability, scalability and isentropicity as we will discuss in the following subsections.

A. Stability

As discussed in [4] and Section II-A for the software implementation transformation, any design transformation can be studied in terms of *stability*. This means that a bounded set of additional functional requirements results only in a bounded set of additional primitives and/or new versions of primitives. As elaborated by Mannaert et al. [4], this would require the absence of so-called combinatorial effects, which result in an impact of additional functional requirements that is proportional to the size of the system. An example of an unstable requirement is for instance a small software application for some recreational sports club that needs to become highly secure and reliable, requiring a completely new and different implementation. In a more traditional engineering context, one could consider the extension of an existing building with additional rooms, which could result in many modifications or even the complete replacement of the plumbing, central heating system, etcetera, of the building.

B. Scalability

The concept of scalability is at least related, and could even be considered to be a special case of stability in this context. Scalability would mean that the increase in value of an existing functional requirement has a clearly defined and limited impact on the constructional view. An example of such a scalable requirement is the amount of concurrent users of a website, which can normally be achieved by adding one or more additional servers. An obvious example of an unscalable requirement in a traditional engineering design is the increase in the number of passengers for an airplane, as this would currently lead to the design of a completely new airplane. In a similar way, an increase of the target velocity or payload capacity of a rocket, generally leads to the design of a completely new and different rocket. One could note here that new rocket manufacturers are indeed trying to scale up existing rocket designs, using more engines or even complete stages in parallel for larger rockets.

C. Isentropicity

With regard to the diagnosability, the concept of isentropicity from statistical thermodynamics can be applied, meaning that an externally observable system state (i.e., a systems macrostate) should completely and unambiguously determine the states of each of the various constituting subsystems (i.e., the systems microstate).

In our view, an isentropic design would therefore imply that the externally observable state of $SysA$ completely and unambiguously determines the states of the various subsystems. An example of such an isentropic design is a finite state machine where the various registers can be read. Indeed, the inputs and register values that are externally observable completely define the internal state of the finite state machine.

VI. Towards the Application of Normalized Systems to Enterprise Engineering

In Sections I and III we argued that not only software applications can be regarded as modular systems, but also many other types of artifacts, such as (for example) organizations. Hence, Sections IV and V focused on a first attempt to extend NS theory concepts to modularity and the systems engineering process in general. In this section, by means of example, we will illustrate some of the implications of our proposed engineering approach when applied to organizational systems. Indeed, several authors have argued for the need of the emergence of an Enterprise Engineering discipline, considering organizations as (modular) systems,

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

which can be 'designed' and 'engineered' towards specific criteria [30], [31], such as (for example) evolvability. More specifically, we will first focus our efforts here at illustrating how the concepts of stability and entropy might offer interesting perspectives to analyze the modular structure of business processes. Next, we will elaborate on the complete and unambiguous definition of organizational modules, as this is in our view a necessary condition to be able to study and optimize the functional/constructional transformation at a later stage.

A. Normalized Systems and Business Process Analysis

When applying the concept of *stability* as considered in NS to business topics, one could consider both business process flows and enterprise architectures. Focusing on business processes, one way to interpret a business process combinatorial effect is the situation in which a single (business process) requirement change leads to N changes in the design of the considered business processes in the business process repository [8].

Consider for instance the handling of a payment incorporating several distinct tasks such as the receiving of an invoice, balance checking, payment execution at the correct date and in the requested format, while taking care of the required accounting and security procedures, etcetera. Suppose that this business process is not separately contained into a single and distinct business process, but instead all kinds of variants are incorporated in all business processes needing to perform payments to (for example) employees, suppliers, moneylenders, and so on. Now further suppose that, for example, a change in legislation would enforce an additional check to be performed for each payment, or a new available payment method would arise. These kind of functional changes would imply an impact of N constructional changes all over the business process repository (i.e., depending on the size of the repository and the number of business processes in which the payment functionality was incorporated). In addition, the precise locations of those modifications within the process repository are unknown upfront and whitebox inspection of each and every process is required to perform the change in a consistent way.

When applying the concept of *entropy* to the level of business process analysis, as we discussed in [12], a first effort should be directed towards interpreting macro- and microstates in such context. Hence, regarding the *macrostate*, typical externally observable properties of a business process, in our view, might include:

- *throughput or cycle time* (how long did the process take to be executed?);
- *quality* and other output related measures (e.g., succesful or non-succesful completion of the process as a whole or the number of defects detected after the execution of the process);
- *costs* involved in the process;
- other *resources* consumed by the process (such as raw materials, electricity, human resources, etcetera).

A typical *microstate*, related to the above sketched macrostate, might then comprise the throughput time of a single task in the process, the correct or erroneous outcome of a single task, the costs related to one activity or the resources consumed by one particular task of the considered business process. Analyzing instantiated business processes in terms of these defined macro- and microstates would then come down to management questions such as:

- which task or tasks in the business process was (were) reponsible for the extremely slow (fast) completion of (this particular instance of) the business process? ;
- which task or tasks in the business process was (were) responsible for the failure of the considered instantiated business process? ;
- which activities contributed substantially or only marginally to the overall cost or resource consumption of the considered business process (cf. cost-accounting and management approaches like Activity Based Costing)?

In case the answer to these questions is unambiguous and clear, the entropy in the system (here: business process repository) is low (or ideally zero) as a particular macrostate (e.g., the extremely long throughput time) can be related to only one or a few microstates (e.g., activity X took three times the normal duration to be carried out, whereas all other activities finished in their regular time span). On the other hand, when no direct answer to these questions can be found, entropy increases: multiple microstates (e.g., prolonged execution of activities X and/or Y and/or Z) could have resulted in the observed and possibly problematic macrostate (e.g., the lengthy execution of the overall process). While we are definitively not the first aiming to relate business processes to entropy (see e.g., [32], [33], [34]), our approach differs in the sense that we consider entropy from the thermodynamics perspective in a run-time environment. An important implication thereof is the fact that entropy in business processes seems to be related to the unstructured aggregation of information and data [12], which could offer interesting research opportunities in (for instance) the accounting domain [35], [36].

B. Towards a Complete and Unambiguous Definition of Organizational Modules

When also considering modules at the organizational level, a considerable effort should equally be aimed at exhaustively listing the interface, incorporating each of its interactions with the environment. This because not including certain dimensions in the interface might evaporate optimization efforts based on stability or entropy reasoning (see Section IV-B and Figure 1(c)).

For instance, when focusing on a payment module, not only the typical 'functional' or 'operational' interface such

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

as the account number of the payer and the payee, the amount and date due, etcetera (typical 'arguments'), but also the more 'configuration' or 'administration' directed interface including the network connection, the personnel needed, etcetera (typical 'parameters') should be included. As such, we might distinguish two kinds of interfaces:

- a *usage interface*: addressing the typical functional and operational (business-oriented) arguments needed to work with the module;
- a *deployment interface*: addressing the typical non-functional, meta-transaction, configuration, administration, ... aspects of an interface.

Although some might argue that this distinction may seem rather artificial and not completely mutually exclusive, we believe that the differences between them illustrate our rationale for a completely defined interface clearly.

While the work of Van Nuffel [8] has resulted in a significant contribution regarding the identification and separation of distinct business processes, the mentioned interfaces still have the tendency to remain underspecified in the sense that they only define the functional 'business-meaning' content of the module but not the other dimensions of the interface, required to fully use a module in blackbox fashion. Such typical other (additional) dimensions — each illustrated by means of an imaginary organizational payment module — might include:

1) Supporting technologies: Modules performing certain functionality might need or use particular external technologies or frameworks. For example, electronical payments in businesses are frequently performed by employing external technologies such as a SWIFT connection or Isabel. In such a case, a payment module should not only be able to interact with these technologies, but the organization should equally have a valid subscription to these services (if necessary) and might even need access to other external technologies to support the services (e.g., the Internet). An organization wanting to implement a module in a blackbox way should thus be aware of any needed technologies for that module, preferably by means of its interface and without whitebox inspection. Suppose that one day, the technology a module is relying on, undergoes some (significant) changes resulting in a different API (application programming interface). Most likely, this would imply that the module itself has to adapt in order to remain working properly. In case the organization has maintained clear and precise interfaces for each of its modules, it is rather easy to track down each of the modules affected by this technological change, as every module mentioning the particular technology in its interface will be impacted. In case the organization has no exhaustively formulated interfaces, the impact of technological changes is simply not known: in order to perform a confident impact analysis, the organization will have to inspect each of the implemented modules with regard to the affected technology in a whitebox way. Hence, technological dependencies should be mentioned explicitly in a module's interface to allow true blackbox (re)use.

2) Knowledge, skills and competences: Focusing on organizations, human actors clearly have to be taken into account, as people can bring important knowledge into an organization and use it to perform certain tasks (i.e., skills and competences). As such, when trying to describe the interface of an organizational module in an exhaustive way, the required knowledge and skills needed for instantiating the module should be made explicit. Imagine a payment module incorporating the decision of what to do when the account of the payer turns out to be insolvent. Besides the specific authority to take the decision, the responsible person should be able (i.e., have the required knowledge and skills) to perform the necessary tasks in order to make a qualitative judgment. Hence, when an organization wishes to implement a certain module in a blackbox way, it should be knowledgeable (by its interface) about the knowledge and skills required for the module to be operational. Alternatively, when a person with certain knowledge or skills leaves the company, the organization would be able to note immediately the impact of this knowledge-gap on the well-functioning of certain modules and could take appropriate actions if needed.

3) Money and financial resources: Certain modules might impose certain financial requirements. For example, in case an organization wants to perform payments by means of a particular payment service (e.g., SWIFT or Isabel), a fixed fee for each payment transaction might be charged by the service company. If the goal is to really map an exhaustive interface of a module, it might be useful to mention any specific costs involved in the execution of a module. That way, if an organization wants to deploy a certain module in a blackbox way, it may be informed about the costs involved with the module ex-ante. Also, when the financial situation of an organization becomes for instance too tight, it might conclude that it is not able any longer to perform the functions of this module as is and some modifications are required.

4) Human resources, personnel and time: Certain processes require the time and dedication of a certain amount of people, possibly concurrently. For example, in case of an organizational payment module, a full time person might be required to enter all payment transactions in the information system and to do regular manual follow-ups and checking of the transactions. As such, an exhaustive interface should incorporate the personnel requirements of a module. That way, before implementing a certain module, the organization is aware of the amount of human resources needed (e.g., in terms of full time equivalents) to employ the module. Equivalently, when the organization experiences a significant decline or turnover in personnel, it might come to the conclusion that it is no longer able to maintain (a) certain module(s) in the current way. Obviously, this dimension is

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

tightly intertwined with the previously discussed knowledge and skills dimension.

5) Infrastructure: Certain modules might require some sort of infrastructure (e.g., offices, materials, machines) in order to function properly. Again, this should be taken into account in an exhaustive interface. While doing so, an organization adopting a particular module knows upfront which infrastructure is needed and when a certain infrastructural facility is changed or removed, the organization might immediately evaluate whether this event impacts the concerning module and vice versa.

6) Other modules or information: Certain modules might use other modules in order to perform their function. For example, when an organization decides to perform the procurement of a certain good, it will probably receive an invoice later on with a request for payment. While the follow-up of a procurement order might be designed into one module, it is reasonable to assume that the payment is designed in a distinct module, as this functionality might also return in other business functions (e.g., the regular payment of a loan). As such, when an organization is planning to implement the procurement module, it should be aware that also a payment module has to be present in the organization to finalize procurements properly. Hence, all linkages and interactions with other modules should be made explicit in the module's interface. When a module (including its interface), used by other modules, is changed at a certain point in time, the adopting organization then immediately knows the location of impact in terms of implemented modules and hence where remedial actions might be required.

In terms of entropy reasoning, the lacking of one or multiple of these above-mentioned dimensions in the organizational module interface can hamper traceability (and hence, diagnosability) regarding the eventually produced outcomes of the organization. For instance, a prolonged throughput time of the payment process can be due to a delay in the communication with one of its external modules or due to the lack of extra knowledge required for the incorporation of an additional legally required check. In case these dimensions are not listed in the module interface, they would typically not be considered as possible causes for the observed result. However, it seems reasonable to assume that an enterprise engineer or dianostician dóes want to be aware of all these possibly problem causing dimensions.

Obviously, it is clear that exhaustively defining the technology, knowledge, financial resources, etc. on which a module depends, will not suffice to solve any of the existing coupling or dependencies among modules. Also, one should always take into consideration that a certain amount of 'coupling' will always be needed in order to realistically perform business functions. However, when the interface of each module is clearly defined, the user or designer is at least aware of the existing dependencies and instances of coupling, knows that ripple-effects will occur if changes affect some of the module's interfaces (i.e., impact analysis) and can perform his or her design decisions in a more informed way, i.e., by taking the interface with its formulated dependencies into account. Consequently, once all forms of hidden coupling are revealed, finetuning and genuine engineering of the concerned modules (e.g., towards low intermodular coupling) seems both more realistic and feasible in a following phase. Indeed, one might deduct that Baldwin and Clark, while defining a module as consisting of powerfully connected structural elements, actually implicitly assumed the existence of an exhaustive set of formulated dependencies before modularization can occur, witness the fact for example that they elaborately discuss Design Structure Matrices [3, chapters 2 & 3]. Our conceptualization is then not to be interpreted as being in contradiction with that of Baldwin and Clark, rather we emphasize more explicitly that the mapping of intermodular dependencies is not to be deemed negligible or self-evident.

VII. On the Feasibility of Applying NS Enterprise Engineering: Some Illustrating Case Studies

In the previous sections, our arguments were mainly illustrated by referring to conceptual examples, such as a tea mug design or imaginary payment module. In this section, we will discuss two short real-life cases further demonstrating the feasibility of how our discussed concepts and reasoning can be applied for analyzing and optimizing realistic organizational problems. Both cases are based on data collected at the administrative department of the authors' research institution and university.

A. Case 1

Our first case analyzes the procedures for the registration of the examination marks of professors and teaching assistants at the end of each semester. In the initial situation, professors and teaching assistants (the 'examiners') were asked to send their grades for each course to the administrative department in one of two possible ways: (1) manually handing in a list of students and their corresponding marks at the secretariat, or (2) sending a mail with a similar list, usually by means of a spreadsheet document. In a next stage, this information would be manually processed by people at the administrative department into the corresponding software applications, generating the student reports afterwards. From the personnel in the administrative department, this processing step required a considerable amount of effort, which kept increasing due to the rising number of students. Hence, an initiative was launched to optimize the way in which the grade administration was performed, aiming to lower the workload on the administrative department. A license for a new software application was bought for this purpose, which was deemed to allow professors and teaching

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

assistants to enter their marks autonomously into the overarching information system by means of a web interface. This way, no direct intervention of the administrative personnel would be required any longer and the students reports would be able to be generated automatically.

Based on our modularity and (NS) enterprise engineering reasoning as discussed above, several remarks can be made. First, as the task 'reporting on the course grades' was embedded in the responsibility of each 'examiner', a necessary 'instability' was noticeable in the form of a *combinatorial effect*. Indeed, the new organizational way of working required all 'examiners' to adapt their individual way of working: no hard-copy forms or direct mails to the secretariat were allowed any longer to register their marks. Instead, as an additional effort compared to the initial situation, each 'examiner' was required to install the correct Internet browser, familiarize himself with the electronic platform, understand the new GUI of the software application, etcetera. One can identify this phenomenon as a combinatorial effect as the considered change (here: the transition to the new application allowing for the web interface) has an impact related to size of the system to which the change has been applied (here: the university having N examining professors and teaching assistants). Such combinatorial effects also have their baleful influence at the organizational performance, such as an increased implementation time of the process optimization directive (i.e., the operation could only succeed after all 'examiners' performed the necessary changes), or an increased risk regarding the incorrect implementation of the directive (i.e., during the implementation efforts of each examiner, errors or inconsistencies might occur).

Second, the new way of working implied a significantly more *complex interface* for all 'examiners' to complete their reporting duties towards the administrative department. In the initial situation, the task 'reporting on the course grades' had an interface, which simply consisted of a plain list with student names and their corresponding grades. In the new situation, this interface became much more complex. Indeed, for successfully handling the task 'reporting on the course grades', the 'examiners' were not only required to provide a list with student results. Instead, 'examiners' were forced to deal with additional concerns on four levels. First, as a 'one-time set-up', the 'examiners' needed to be able to install the correct Internet browser (version) on their computer to be able to access the web interface of the new software application and correctly set-up a VPN client to allow for a secured connection with the university's network. Next, as a 'pre-operation set-up', the 'examiners' needed to perform the correct log-in procedures each time they wanted to report on some course grades, to actually establish the secure network. Third, some 'general competences' were required from the 'examiners' to be able to interact with the browser, the web application (including the new and rather complex GUI) and VPN application, understand the web application's specific coding scheme (e.g., to indicate that a student was legitimately absent), etcetera. Fourth, in case problems arose regarding any of these issues, the 'examiners' were implicitly believed to be able to provide the correct fault and 'exception handling' for all these issues (e.g., dealing with validation errors, anomalies, strange menus, . . .). Consequently, to correctly comply with the new interface, 'examiners' were essentially forced to deal with concerns regarding the (sometimes technical) complexity of the new software application, whereas in the initial situation, people from the administrative personnel were the only ones required to interface with such software application. Hence, in the initial situation, only people from the administrative department needed to have knowledge of these specificities. However, given the more complex interface in the new way of working, many professors and teaching assistants were required to ask for assistance from the administrative personnel due to many problems of often different origins, as they had insufficient knowledge to resolve them independently (i.e., the knowledge implicitly deemed to be present, was not always available).

As a consequence, it was strikingly to note that some people at the administrative department reported that the time and effort required to assist these people, was perceived as exceeding the original effort to manually process all examination results themselves (as was the case in the initial situation). Again, unanticipated baleful consequences for the organizational performance were noticeable. This was clearly not the desired result, as the initial goal was precisely to reduce the overall administrative efforts significantly. Obviously, we do not want to claim in this analysis that automatization efforts are to be deemed counterproductive per se. Rather, we want to illustrate how some ways of working might result in instability issues or problems related to an underspecified or complicated modular interface. For instance, it was noticed that other kinds of 'self-service' automatization efforts (e.g., a new web application allowing students to subscribe themselves and choose their individual course program for each academic year) resulted in very similar problems. In essence, a similar situation arose in which many concerns (deemed to be trivial to deal with for all people) were exported from a centralized department to all people involved in the self-service. In case these concerns however turn out to be non-trivial later on, many problems occur as their complexities are pushed to all involved people via the new interface.

B. Case 2

Our second case focuses on the consequences of a change in the communication policy, such as an adapted company logo and style. Historically, the considered case organization originated from the merge of three existing universities into one larger university. One implication of this organizational

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

merge was the need to 'market' the new university by (amongst others) introducing a new company logo and style (together with the new and correct contact details, VAT-numbers, etcetera), incorporated on (for instance) all official letters (i.e., by means of the printings on 'stationery' or university paper). The traditional routine for sending letters within the organization allowed each of the more than 4000 university members (the 'senders') to autonomously print and send letters by using the 'old' letter style (i.e., on university paper). Analyzing the considered change in the communication policy, we find again a rather large and complex impact of such a (seemingly) small organizational change.

First, as each 'sender' is responsible for taking care of the concern of applying the correct letter lay-out, a change regarding this company logo and style results in a *combinatorial effect*. Indeed, to correctly implement the new company style, each 'sender' should adapt his own way of working, by using and applying the new company logo and style (i.e., confirming to and printing on the new university papers), and hold back from using the old stationery. Hence, the impact of the applied change (here: introducing the new company logo) is dependent on the size of the target system (here: the N 'sending' employees). Also here we could identify additional baleful effects regarding the performance of the organization. One aspect involves again the increased implementation time of the change and the increased risk regarding the correct implementation of the directive (i.e., each and every 'sender' has to adapt his individual way of working and do so in a correct way). Another aspect concerns the tendency of individual university members to 'pile up' an individual 'stock' of stationery on his or her desk due to perceived efficiency reasons. Clearly, this causes a significant loss of stock, proportional to the number of university members, from the moment the change in communication policy has been announced, as the old stock of stationery becomes obsolete.

Additionally, in *entropy* reasoning, some findings (i.e., a macrostate) may suggest that some people seem not to apply the new 'rules' regarding the company logo and lay-out (e.g., given the fact that some clients still employ old VAT-numbers based on recently sent letters printed on and old version of the university paper). However, as no control or logging is kept on who is sending letters on which moment, no clear diagnosis can be made as to who (i.e., which 'sender') is still reluctant to applying the new directive. Indeed, each person sending letters after the new directive was valid, could have sent letters with a wrong letter lay-out and contact details (i.e., multiple microstates) and uncertainty arises.

A possible optimization in the organizational design could for example consist of a situation wherein each person who wants to send a letter, only drafts the letter in terms of its content and afterwards sends this 'e-letter' to the administrative department, which prints the letter on the correct version of the stationery, having the appropriate header and contact details. In such case, the combinatorial effect would be eliminated. Changes regarding the lay-out, logo or general contact information on letters would have an impact limited to the personnel of the administrative department, while the tasks carried out by people who 'send' letters remains unchanged (i.e., only having responsibility regarding the content of letters) and are change independent regarding this concern. Also, in case an error would nevertheless occur, less uncertainty would be present regarding who might have applied the wrong lay-out, as only people from the administrative department are entitled to print-out letters. Finally, also improvement on the organization's performance could be expected: changes in the communication policy would be applied in a more easy, fast and correct way, and no individual stocks of the stationery would be present any longer.

Consequently, the applications of our NS Enterprise Engineering approach to these cases, might show how our reasoning may lead to real-life and relevant organizational problem solving and decision making.

VIII. Interpreting Entropy and Instability as Coupling within Modular Structures

In the previous sections, we elaborated on how NS uses the concepts of stability and entropy to design and optimize the modular structure of software applications and how this reasoning can be generalized to modular systems in general, such as for example organizational systems. In this section, we will take a more broad perspective, reflecting on the essential meaning of stability and entropy in this modularity approach. While in essence, both stability (cf. systems theory) and entropy (cf. thermodynamics) have their origin in distinct theories, we will argue that one way to interpret both concepts is to consider them as pinpointing to other 'symptoms' of the same underlying 'cause' or phenomenon, being the coupling (or interaction) between particles and subsystems of a larger system. In advance, it should be emphasized that the following discussion has a rather explorative, reflective and tentative goal, rather than fully formalizing each aspect. However, given the earlier elaborate discussion and reasoning regarding modularity, coupling, stability and entropy, the authors believe that the given interpretation might clarify some of the claims presented above.

A. Stability and coupling

In fact, the concept of stability from systems theory might be interpreted as referring to *coupling regarding the design parameters or dimensions of modular structures*. Consider for instance a standard example of dynamic (in)stability analysis in traditional engineering: the construction of a bridge. Typically, such buildings or construction 'systems'

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

have a natural frequency of vibration (a so-called 'eigenfrequency'). In case the frequency of the oscillation to which the system is exposed matches this eigenfrequency (e.g., due to gusts of wind), the system absorbs more energy than it does in case of any other frequency: the bridge may be forced into violent swaying motions, leading to mechanical damages and ultimately sometimes even leading to its destruction. In such a case, the system absorbs and aggregates or accumulates the energy throughout its whole structure causing an extreme 'output' instead of the gradual reduction of the wave as it occurs at all other frequencies. Hence, this transformation process can be considered to be instable as a bounded input causes an unbounded and uncontrolled output.

Consequently, engineers of such buildings have to be aware of this coupling in terms of design parameters (e.g., between the buildings eigenfrequency and the frequency of frequently occurring gusts of wind) when devising the system in order to avoid such instability effects. In terms of our modularity approach discussed above, we would require to incorporate this dependency of the 'building system' on the external environment into its 'interface'. Only then could one be able to regard the building as a genuine 'module' as the dependency reveals one type of interaction between the considered system (i.e., the building) and the environment it operates in (i.e., the meteorological conditions). Typically, engineers will try to decouple (i.e., fragment) both aspects by incorporating for example shock mounts to absorb the resonant frequency and compensate (i.e., cancel) the resonance and absorbed energy. Alternatively, the engineers might choose to design the construction in such a way that the building only resonates at certain frequencies not typically occurring. In reality, a combination of both practices will most likely be opted for. Consequently, these remedial measures can be regarded as corresponding to the optimization of the modular arrangements and their interfaces towards a specific criterion (here: the avoidance or reduction of mechanical resonance).

One thing the engineer should definitely avoid at all times is hidden coupling in this respect: the case in which one is not aware of this interaction (i.e., coupling) in terms of design parameters. In the discussed example, not being aware of the eigenfrequency and the danger it implies in terms of instable reactions, might obviously be disastrous. Hence, the coupling of the subsystem with regard to the overall system it operates in, should be mentioned in the interface to safely regard the module as a blackbox. If not all kinds of coupling are documented, whitebox inspection will still be needed to assess the impact of external changes (in our example: a gust of wind).

Another example for illustrating our interpretation of instability in terms of coupling can be found by considering violations towards NS theorems. Consider for instance the *Action version Transparency* theorem. Imagine an action entity A not exhibiting version transparency and being confronted with a mandatory version upgrade: each action entity calling this (new version) of action entity A would then be required to be adapted in terms of its call towards A. As such, employing the assumption of unlimited systems evolution, the impact of this external change can become unbounded or unstable. Also here, it can be clearly seen that the instability is caused by means of coupling within the modular structure in terms of its design parameters. As each action entity calling action entity A has a piece of implemented software code depending on the specificities of action entity A, action entity A is coupled (and interacting) with all of its calling entities through its technical design. Here again, the software engineer would preferably opt for, first, recognizing this dependency (i.e., including it in its interface) and, in a second stage, introducing fragmentation in order to control the coupling. This can for instance be done by employing action entities, which do allow for Action version Transparency, hence isolating each of the considered concerns for this dimension.

More generally, each of the NS theorems from the stability point of view (i.e., *SoC*, *AvT*, *DvT* and *SoS*) can be interpreted as theorems enforcing the decoupling of aspects causing instabilities in the modular design in case they would not be separated. Indeed, *Separation of Concerns* enforces all change drivers to be separated in the design of a software architecture in order to have modules dependent on only one independent concern, not coupled to other change drivers. *Separation of States* decouples various modules in the sense that new (error) states do not get escalated in the design towards other modules. *Action version Transparency* and *Data version Transparency* even precisely aim at the more 'traditional' interface between modules, when demanding that new versions should have an interface not impacting already existing data and action entities.

B. Entropy and coupling

When considering entropy as defined in (statistical) thermodynamics, this concept might be interpreted as referring to *coupling between subparts or particles of a modular system in terms of their run-time execution*. In fact, one way to understand entropy is the natural tendency of particles in a system to interact (see [12] and Section II-B). Indeed, the fact that an increased amount of entropy is associated with an increasing number of possible microstate configurations consistent with one macrostate, can essentially be traced back to the uncontrolled interaction between these particles. Consider for instance again the gas containment illustration as depicted in Figure 1. We know that if the gas is contained in only one out of the two components in the container, entropy will increase as soon as the shaft between the two components is removed: both components become one single space, and the gas will dissipate and spread out into the full space, interacting with the parts in the second

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

component of the container throughout time. Due to the removal of the shaft, both components become '*coupled*' in their run-time dimension and will be subject to their natural tendency to interact with each other throughout time. This interaction (and the removal of the fragmentation or structure within the space) increases the degree of entropy as now a higher number of microstates (configurations of the molecules) can result in a single macrostate.

The only way to avoid such entropy increase, is by introducing structure or fragmentation in this run-time dimension of the modular structure. In our container example, this might be done by maintaining the shaft in the container and hence avoiding additional interaction between the molecules. As repeatedly mentioned before, it is important to note that this additional structure (such as an interface) needs to be complete and exhaustive. This can again be nicely illustrated by Figure 1, Panel (c). While a part of the shaft is still in place and possibly aimed at avoiding interaction, the interaction between both compartments still takes place as the decoupling or fragmentation is incomplete, and entropy increase will occur.

More generally, each of the NS theorems from the entropy point of view (i.e., *SoC*, *AiT*, *DiT* and *SoS*) can be interpreted as theorems enforcing the decoupling of aspects causing entropy generation in their run-time dimension in case they would not be separated. Indeed, *Separation of States* decouples various modules during the run-time execution of a software application in the sense that the statefull calling will generate a persistent state after completion of each action (and hence, information about the microstate is retained and externalized). *Separation of Concerns* enforces concerns (here: information units) to be separated so that each concern of which independent information should be traceable, is contained in a separate module (and hence, state). *Action instance Transparency* and *Data instance Transparency* even specifically aim at the fact that the versions, values and threads of actions and data need to be logged during each instantiation for traceability or diagnosability purposes.

In conclusion, as we have claimed in this section that one way to interpret both the occurrence of instability and entropy in modular structures is their relation to the existence of uncontrolled coupling, the need for completely and exhaustively defined interfaces becomes even more pertinent if one's goal is to obtain a stable modular structure exhibiting isentropicity. Moreover, structures allowing 'hidden coupling' (i.e., still leaving room for certain forms of leakage in the design or run-time dimension) should be considered harmful or 'misleading' in the sense that the designer might be convinced of the stability or isentropicity of his designed structure, while in reality the resulting module boundaries or states do not reflect an isolated part of the system, fully decoupled from the rest of the system.

IX. Conclusion and Future Work

This paper focused on the further exploration and generalization of NS systems engineering concepts to the analysis and design of modules in systems in general, and organizational systems in particular. The current state-of-the-art regarding NS and modularity was reviewed, primarily focusing on the seminal work of Baldwin and Clark. Subsequently, we argued that, first, a distinction should be made between blackbox (functional) and whitebox (constructional) perspectives of systems. As the practical blackbox (re)use of modules requires the absence of any hidden coupling, the need for complete and exhaustively defined interfaces was argued for. Next, a discussion of some properties of the functional/constructional transformation was proposed. As we believe that this systems engineering approach might be useful for optimizing other modular structures (including organizations) as well, we discussed some of the implications of this reasoning when applied to Enterprise Engineering (such as stability, entropy and complete interfaces) and provided two short real-life cases for illustrative purposes. We concluded that our conceptualization of is not in contradiction with that of Baldwin and Clark, but rather emphasizes an additional intermediate design stage when devising (organizational) modules. Also, from a broader modularity viewpoint, we provided an interpretation of the traditional engineering concepts of instability and entropy as being manifestations of coupling in modular structures (regarding their design or run-time dimensions, respectively).

Regarding our applications towards Enterprise Engineering, a limitation of this paper can be seen in the fact that no guarantee is offered that the identified additional interface dimensions will reveal all kinds of hidden coupling in every organization. Therefore, additional research (e.g., extra case studies) with regard to possible missing dimensions seems to be required. In addition, our application of modularity and NS concepts to the organizational level was limited to the definition of completely defined organizational modules and the illustration of the existence of instability and entropy generation at a business process level. A completely defined stable and isentropic functional/constructional transformation on the organizational level (as it exists on the software level) was still out of scope in this paper. Furthermore, future research at our research group will be aimed at identifying and validating organizational blackbox reusable modules, exhibiting exhaustively defined interfaces and enabling the bottom-up functional/constructional transformation.

Regarding the discussion of our interpretation of entropy and stability as different manifestations of coupling, it should be noted that it is to be considered as a mainly exploratory effort, needing further formalization in future research.

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

ACKNOWLEDGMENT

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

REFERENCES

[1] P. De Bruyn and H. Mannaert, "Towards applying normalized systems concepts to modularity and the systems engineering process," in *Proceedings of the Seventh International Conference on Systems (ICONS) 2012*, 2012, pp. 59 – 66.

[2] C. Y. Baldwin and K. B. Clark, "Managing in an age of modularity," *Harvard Business Review*, vol. 75, no. 5, pp. 84–93, 1997.

[3] ——, *Design Rules: The Power of Modularity*. Cambridge, MA, USA: MIT Press, 2000.

[4] 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, special Issue on Software Evolution, Adaptability and Variability.

[5] ——, "Towards evolvable software architectures based on systems theoretic stability," *Software Practice and Experience*, vol. Early View, 2011.

[6] H. Mannaert, P. De Bruyn, and J. Verelst, "Exploring entropy in software systems: towards a precise definition and design rules," in *Proceedings of the Seventh International Conference on Systems (ICONS) 2012*, 2012, pp. 93 – 99.

[7] D. Van Nuffel, H. Mannaert, C. De Backer, and J. Verelst, "Towards a deterministic business process modeling method based on normalized systems theory," *International Journal on Advances in Software*, vol. 3, no. 1-2, pp. 54–69, 2010.

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

[9] P. Huysmans, "On the feasibility of normalized enterprises: Applying normalized systems theory on the high-level design of enterprises," Ph.D. dissertation, University of Antwerp, 2011.

[10] H. Mannaert and J. Verelst, *Normalized systems: re-creating information technology based on laws for software evolvability*. Koppa, 2009.

[11] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060 – 1076, sept. 1980.

[12] P. De Bruyn, P. Huysmans, G. Oorts, and H. Mannaert, "On the applicability of the notion of entropy for business process analysis," in *Proceedings of the Second International Symposium on Business Modeling and Software Design (BMSD) 2012*, 2012, pp. 128 – 137.

[13] G. Anderson, *Thermodynamics of Natural Systems*. Cambridge University Press, 2005.

[14] C. Shannon, "A mathemathical theory of communication," *Bell System Technical Journal*, vol. 28, pp. 379–423, 1948.

[15] H. Leff, "Thermodynamic entropy: The spreading and sharing of energy," *American Journal of Physics*, vol. 64, no. 10, pp. 1261–1271, Oct 1996.

[16] L. Boltzmann, *Lectures on gas theory*. Dover Publications, 1995.

[17] D. Steward, "The design structure system: A method for managing the design of complex systems," *IEEE Transactions in Engineering Management*, vol. 3, no. 28, pp. 71 – 84, 1981.

[18] S. Eppinger, "Model-based approaches to managing concurrent engineering," *Journal of Engineering Design*, vol. 2, no. 4, pp. 283 – 290, 1991.

[19] H. Simon, *The Sciences of the Artificial*, 3rd ed. Cambridge, Massachusetts: MIT Press, 1996.

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

[21] H. Simon, "The architecture of complexity," in *Proceedings of the American Philosophical Society*, vol. 106, no. 6, December 1962.

[22] R. Sanchez and J. Mahoney, "Modularity, flexibility, and knowledge management in product and organization design," *Strategic Management Journal*, vol. 17, pp. 63–76, 1996.

[23] M. Op't Land, "Applying architecture and ontology to the splitting and allying of enterprises," Ph.D. dissertation, Technical University of Delft (NL), 2008.

[24] G. M. Weinberg, *An Introduction to General Systems Thinking*. Wiley-Interscience, 1975.

[25] L. Bertalanffy, *General Systems Theory: Foundations, Development, Applications*. New York: George Braziller, 1968.

[26] M. Bunge, *Treatise on Basic Philosophy: Vol. 4: Ontology II: A World of Systems*. Boston: Reidel, 1979.

[27] J. Rowson and A. Sangiovanni-Vincentelli, "Interface-based design," in *Proceedings of the 34th Design Automation Conference*, 1997.

[28] E. Dijkstra, "Go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[29] P. De Bruyn, D. Van Nuffel, P. Huysmans, and H. Mannaert, "Towards functional and constructional perspectives on business process patterns," in *Proceedings of the Sixth International Conference on Software Engineering Advances (ICSEA)*, Barcelona, Spain, 2011, pp. 459–464.

[30] J. L. G. Dietz, *Enterprise Ontology: Theory and Methodology*. Springer, 2006.

[31] J. Hoogervorst, *Enterprise Governance and Enterprise Engineering*. Springer, 2009.

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[32] J.-Y. Jung, "Measuring entropy in business process models," in *3rd International Conference on Innovative Computing Information and Control, 2008 (ICICIC '08)*, june 2008.

[33] J.-Y. Jung, C.-H. Chin, and J. Cardoso, "An entropy-based uncertainty measure of process models," *Information Processing Letters*, vol. 111, no. 3, pp. 135 – 141, 2011.

[34] J. Trienekens, R. Kusters, D. Kriek, and P. Siemons, "Entropy based software processes improvement," *Software Quality Journal*, vol. 17, pp. 231–243, 2009, 10.1007/s11219-008-9063-6.

[35] J. Ronen and G. Falk, "Accounting aggregation and the entropy measure: An experimental approach," *The Accounting Review*, vol. 48, no. 4, pp. 696–717, 1973.

[36] A. R. Abdel-Khalik, "The entropy law, accounting data, and relevance to decision-making," *The Accounting Review*, vol. 49, no. 2, pp. 271–283, 1974.

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org