Towards Applying Normalized Systems Concepts to Modularity and the Systems Engineering Process

Peter De Bruyn, Herwig Mannaert
Department of Management Information Systems
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. Modularity has frequently been suggested as a powerful means for reducing that complexity and enabling flexibility. As Normalized Systems (NS) theory has proven to introduce this evolvable modularity in software systems, this paper further explores the generalization of NS systems engineering concepts to modularity and the systems engineering process in general, and organizational systems in particular. After emphasizing the distinction between blackbox and whitebox perspectives on systems, we focus on the importance of employing exhaustively defined interfaces as a prerequisite to obtain 'true' black-box modules. Some aspects of the functional/constructional transformation are discussed. Finally, six additional interface dimensions are proposed as possible aspects to be included in such exhaustive interfaces when considering organizational modularity.**

***Keywords*-Normalized Systems, Modularity, Systems engineering, Evolvability, Systems theoretic stability**

I. INTRODUCTION

Current organizations need to be able to cope with increasing change and increasing complexity in many or all of their aspects. In this regard, modularity has previously been suggested as a powerful means for reducing that complexity by decomposing a system into several subsystems. Moreover, modifications at the level of those subsystems in stead of the system as a whole are said to be facilitating the overall evolvability of the system. More specifically, *Normalized Systems (NS) theory* has recently proven to introduce this evolvable modularity, primarily at the level of software systems. First, 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 $\mathcal{S}$ [1], [2], [3]:

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

Next, in order to obtain evolvable modularity, NS theory states that this transformation should exhibit systems theoretic stability, meaning that a bounded input function (i.e., bounded set of requirement changes) should result in a bounded output values (i.e., a bounded impact or effort) even if an unlimited systems evolution is considered. Furthermore, Mannaert et al. [2] have formally proven that this implies that modular structures should strictly adhere to the following *principles*:

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

As this 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 software *elements*, i.e., action elements, data elements, workflow elements, trigger elements and connector elements [1], [2], [3]. Typical cross-cutting concerns (such as remote access, persistence, access control, etc.) are included in these elements in a way which is consistent with the above-mentioned theorems.

However, 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. Indeed, Van Nuffel has recently shown the feasibility of applying modularity and NS theory concepts at the business process level [4], [5] while Huysmans did so at the level of enterprise architectures [6]. However, the extension of NS theory to these domains has not been formalized yet. Consequently, as NS theory proved to be successful in introducing evolvable modularity in software systems, and as it is clearly desirable to extend such properties to other systems, this paper focuses on a first step towards generalizing NS theory concepts to systems engineering in general. More specifically we will focus on and stress the importance of a complete and unambiguous definition of the interface of modular subsystems as an

 ISBN: 978-1-61208-184-7

essential part of each systems engineering process. Next, by way of illustration, we discuss an initial attempt of applying our approach to organizational systems, motivated by the frequently observed discrepancy between the uttered need to more systematically engineer organizational artifacts [7], [8] and empirical findings suggesting that mostly only ad hoc approaches are employed in practice [9].

The remainder of this paper is structured as follows. Section II will discuss some extant literature on modularity, emphasizing the work of Baldwin and Clark. Next, we propose a more unambiguous definition of modularity after discussing functional and constructional perspectives on systems (Section III) and outlining some of the transformation properties (Section IV). Finally, some exemplifying additional interface dimensions when considering organizational modules will be suggested (Section V), as well as some conclusions and opportunities for future research (Section VI).

II. Related Work

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 [10]. While no single generally accepted definition is known, the concept is most commonly associated with the process of subdividing a system into several subsystems [11], [12]. This decomposition of complex systems is said to result in a certain degree of complexity reduction [13] and facilitate change by allowing modifications at the level of a single subsystem in stead of having to adapt the whole system at once [14], [10], [15].

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*" [15, 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. Consequently, modularization is conceived 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. A set of design rules (visible information) is 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 allows multiple (parallel) experiments for each module 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 [15]:

- *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 phyisical 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 [1]. 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, and even whole industry clusters [15]. Also, Campagnolo and Camuffo [12] investigated the use of modularity concepts within management science and identified 125 studies in which modularity concepts arose as a design principle of organizational complex systems, suggesting that the principles 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, modularity equally proved to be a powerful concept. For instance, Op't Land used modularity related criteria to merge and split organizations [16]. Van Nuffel proposed a framework to deterministically identify and delimit business processes based on a modular and NS theory viewpoint [4], [5], and Huysmans demonstrated the usefulness of modularity with regard to the study of (the evolvability) of enterprise architectures [6].

III. 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 [15], already describes an ideal form of modularity (e.g., loosely coupled and independent). As such, we will first discuss the need to distinguish the functional and constructional perspectives of systems. Next, we will propose to introduce the formulation

 ISBN: 978-1-61208-184-7

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.

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 [17]. The functional perspective focuses on describing what a particular system or unit does or what its function is [18]. 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 structural 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?) [19].

Equivalently, one could regard the functional system view as a blackbox representation, and the constructional system view as a witebox representation. By blackbox 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 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 1: 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, ... 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, ... (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.

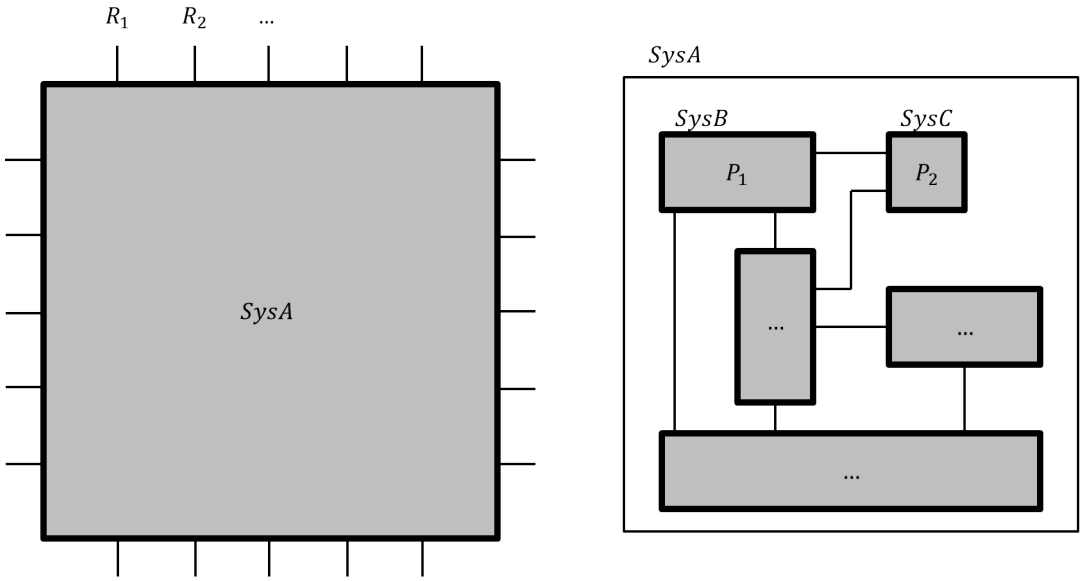
B. Avoiding Hidden Coupling by Strictly Defining Modular Interfaces

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, exclusively relying on the available interface, thus without having any knowledge about the inner way of working of the concerned module. Stated otherwise, the interface of the module should describe any possible dependency regarding the module, needed to perform its functionality. Consequently, every interaction of the system with its environment should be properly and exhaustively defined herein. 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, etc. are consequently also to be taken into account (cf. Section V). Not formulating these 'tacit assumptions' 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 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 black box 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

 ISBN: 978-1-61208-184-7



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

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 furter as they make it possible to combine processing actions and data entities into one primitive (i.e., a single class) [2]. 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.

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, etc. are precisely, exante, 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 characterics 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 II 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, 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.

 ISBN: 978-1-61208-184-7

IV. Towards Applying Systems Engineering on 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 [2], 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.

Stability: As discussed in [2] 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. [2], this would require the absence of so-called combinatorial effects resulting 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 in an "office" environment that needs to become highly secure and reliable, requiring a completely new and different implementation.

Scalability: 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. Examples of unscalable requirements in current designs are the increase in the number of passengers in an airplane, leading to the design of a completely new airplane, or the increase in the target velocity of a rocket, leading to the design of a totally different rocket.

Normalization: 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 [1], [2] for the implementation of elementary functional requirements into software primitives (in this case elements as structured aggregations of primitives).

Isentropicity: Entropy is defined in statistical thermodynamics as the number of microstates for a given macrostate, corresponding to the uncertainty of the detailed internal system state with respect to the observable external state [20], [21]. In our view, an isentropic design would therefore imply that the external 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.

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 [22]. 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.

V. On a Complete and Unambiguous Definition of Organizational Modules

In Sections I and II 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 III and IV 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 which can be 'designed' and 'engineered' towards specific criteria [7], [8], such as (for example) evolvability. More specifically, we will primarily focus our efforts here 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.

Consequently, when also considering modules at the organizational level, a first effort should equally be aimed at exhaustively listing the interface, incorporating each of its interactions. For instance, when focusing on a payment module, not only the typical 'functional' interface such as the account number of the payer and the payee, the amount and date due, etc. (typical 'arguments') but also the more 'configuration' or 'administration' directed interface including the network connection, the personnel needed, etc. (typical 'parameters') should be included. As such, we might distinguish two kinds of interfaces:

- a *usage interface*: addressing the typical functional (business-oriented) arguments needed to work with the module;

 ISBN: 978-1-61208-184-7

- 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 [5] 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 SWIFT 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 tightly intertwined with the previously dicussed 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

 ISBN: 978-1-61208-184-7

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.

Obviously, it is clear that exhaustively defining the technology, knowledge, financial resources, ... 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. 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.

VI. Conclusion and Future Work

This paper focused on the further exploration and generalization of NS systems engineering concepts to modularity and the systems engineering process in general, and organizational systems in particular. The current state-of-the-art regarding 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 and whitebox perspectives of systems. A system can then be considered as the transformation of (functional) requirements into (constructional) primitives. A preliminary discussion of some properties of this transformation was proposed. Next, in order to be able to fully (re)use those constructional primitives as 'blackbox building blocks', we proposed to define a module as a subsystem which can be completely described solely by its interface, thus indicating exhaustively all interactions and dependencies and hence avoiding hidden coupling. Finally, six additional organizational interface dimensions when considering organizational modules were suggested, implied by our approach. We concluded that our conceptualization is not in contradiction with that of Baldwin and Clark, but rather emphasizes an additional intermediate design stage when devising (organizational) modules.

A limitation of this paper is that no garantuee is offered that the identified additional interface dimensions will reveal all kinds of hidden coupling in every organization. Therefore, additional research (e.g., case studies) with regard to possible missing dimensions is required. In addition, our application of modularity and NS concepts to the organizational level was limited to the definition of completely defined organizational modules. The functional/constructional transformation on the organizational level was still out of scope in this paper. Furthermore, future research at our research group will be aimed at identifying and validating organizational black-box reusable modules, exhibiting exhaustively defined interfaces and enabling the bottom-up functional/constructional transformation.

Acknowledgment

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

References

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

[2] 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. Article in press, 2011.

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

[4] 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.

 ISBN: 978-1-61208-184-7

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

[6] 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.

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

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

[9] M. Indulska, J. Recker, M. Rosemann, and P. F. Green, "Business process modeling: Current issues and future challenges," in *CAiSE*, ser. Lecture Notes in Computer Science, P. van Eck, J. Gordijn, and R. Wieringa, Eds., vol. 5565. Springer, 2009, pp. 501–514.

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

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

[12] 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.

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

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

[15] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity*. Cambridge, MA, USA: MIT Press, 2000.

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

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

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

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

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

[21] Wikipedia. (2011) Entropy. [Online]. Available: http://en.wikipedia.org/wiki/Entropy

[22] 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.

 ISBN: 978-1-61208-184-7