On the Variability Dimensions of Normalized Systems Applications: Experiences from an Educational Case Study

Peter De Bruyn, Herwig Mannaert and Philip Huysmans

Department of Management Information Systems
Faculty of Applied Economics
University of Antwerp, Belgium
Email: {peter.debruyn,herwig.mannaert,philip.huysmans}@uantwerp.be

***Abstract*—Normalized Systems Theory (NST) aims to create software systems exhibiting a proven degree of evolvability. While its theorems have been formally proven and several applications have been used in practice, no real overview of the typical types or dimensions along which such NST compliant applications can evolve is present. Therefore, this paper presents an NST case within an educational context in which its different variability dimensions are illustrated. Based on this case, a more general overview of 4 variability dimensions for NST applications is proposed: changes regarding the application model, expanders, craftings and technological options.**

***Keywords*–Evolvability; Normalized Systems; Variability dimensions; Case Study**

I. Introduction

The evolvability of information systems (IS) is considered as an important attribute determining the survival chances of organizations, although it has not yet received much attention within the IS research area [1]. Normalized Systems Theory (NST) was proposed as one theory to provide an ex-ante proven approach to build evolvable software by leveraging concepts from systems theory and statistical thermodynamics [2]–[4]. The main dimensions of evolvability or variability facilitated by the theory have nevertheless not yet been thoroughly documented. Additionally, while some NST cases have been documented in extant literature [5]–[9], the overall number of cases is still fairly limited. This paper attempts to tackle both mentioned gaps by discussing an NST application, which was built and used for the management of process evaluations of master dissertations at the faculty of the authors. Based on this case, we discuss the different dimensions along which variations in an NST application can arise. These dimensions are consequently also important indications with respect to the main areas in which an NST application can evolve throughout time.

The remainder of this paper is structured as follows. In Section II, we briefly present NST as the theoretical basis on which the considered software application was built. Section III provides some general context regarding the case: why and how the application was used. Next, the case is further analyzed in Section IV. We offer a discussion in Section V and our conclusion in Section VI.

II. Normalized Systems Theory

The case application we will present and analyze in the following sections, is based on NST. This theory has been previously formulated with the aim of creating software applications exhibiting a proven amount of evolvability [2]–[4]. More specifically, the goal is to eliminate the generally experienced phenomenon in which software systems become more difficult to maintain and adapt, as they become bigger and evolve throughout time [10].

NST is theoretically founded on the concept of *stability* from systems theory. Here, stability is considered as an essential property of systems. Stability means that a bounded input should result in a bounded output, even if an unlimited time period $T \rightarrow \infty$ is considered. In the context of information systems, this implies that a bounded set of changes should only result in a bounded impact to the system, even in cases where $T \rightarrow \infty$ (i.e., considering an unlimited systems evolution). Put differently, it is demanded that the impact of changes to an information system should not be dependent on the size of the system to which they are applied, but only on the size and property of the changes to be performed. Changes dependent on the size of the system are called *combinatorial effect*. It has been formally proven that any violation of any of the following *theorems* will result in combinatorial effects (thereby hampering evolvability) [2]–[4]:

- *Separation of Concerns*, stating that each concern (i.e., each change driver) needs to be separated from other concerns in its own construct;
- *Action Version Transparency*, stating that an action entity should be able to be updated without impacting the action entities it is called by;
- *Data Version Transparency*, stating that a data entity should be updateable without impacting the action entities it is called by;
- *Separation of States*, stating that all actions in a workflow should be separated by state (i.e., being called in a stateful way).

The application of the theorems in practice has shown to result in very fine-grained modular structures, which are generally considered to be difficult to achieve by manual programming. Therefore, NST proposes five *elements* (action, data, workflow, connector and trigger) that serve as design patterns [3], [4]:

- *data element*: a set of software constructs encapsulating a data construct (including a set of convenience methods, such as get- and set-methods, and providing remote access and persistence), allowing data storage and usage within an NST application;

- *action element*: a set of software constructs encapsulating an action construct (providing remote access, logging and access control), allowing the execution of units of processing functionality within an NST application;
- *workflow element*: a set of software constructs allowing the execution of a sequence of action elements (on a specific data element) within an NST application;
- *connector element*: a set of software constructs enabling the interaction of an NST application with external systems and users in a stateful way;
- *trigger element*: a set of software constructs enabling the triggering of action elements within an NST application, based on error and non-error states.

Based on these elements, NST software is generated in a relatively straightforward way through the use of the *NST expansion mechanism*. First, a model of the considered universe of discussion is defined in terms of a set of data, action and workflow elements. Next, NST expanders generate parameterized copies of the general element design patterns into boiler plate source code. Several layers can be discerned in this code: a shared layer (not containing any reference to external technologies), data layer (taking care of data services), logic layer (taking care of business logic and transactions), remote or proxy layer (taking care of remote access), control layer (taking care of the routing of incoming requests to the appropriate method in the appropriate class in the proxy layer) and view layer (taking care of presenting the view to be rendered by the user interface, such as a web browser). This generated code can, if preferred, be complemented with *craftings* (custom code) to add non-standard functionality that is not provided by the expanders themselves at well specified places (anchors) within the boiler plate code. The boiler plate code together with the optional craftings are then compiled (built) so that the application can be deployed.

III. CASE INTRODUCTION

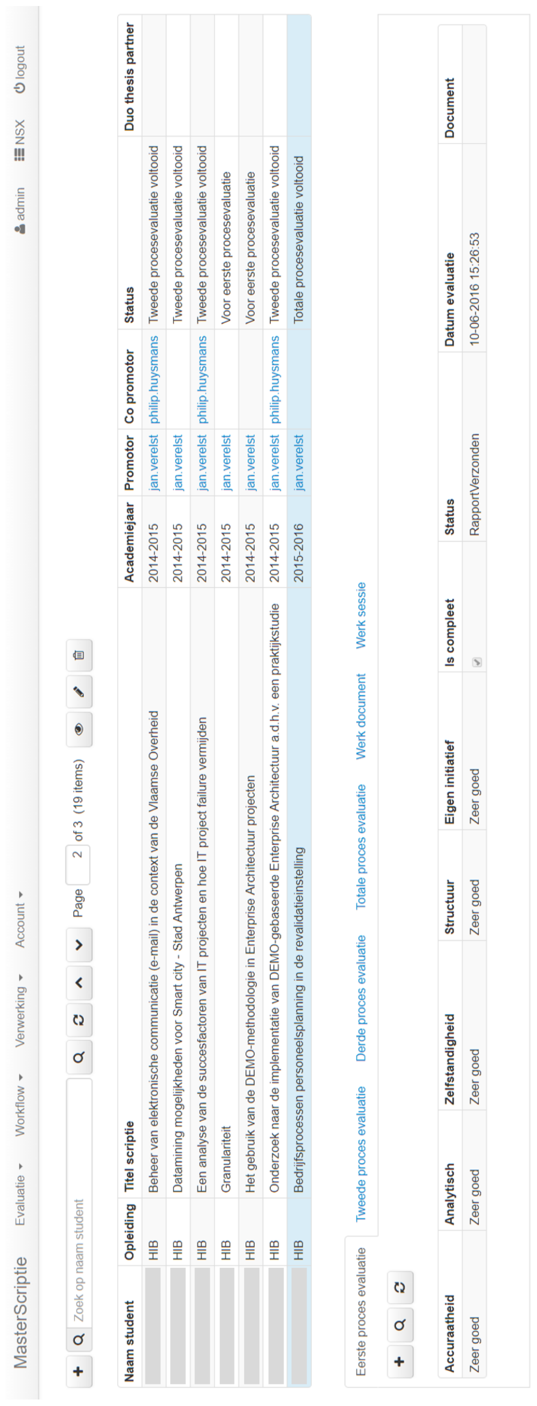
The case we present is concerned with the master thesis evaluations at the Faculty of Applied Economics of the University of Antwerp. At the university, master students writing their dissertation are not only evaluated with regard to the end result (i.e., the thesis itself) but also (for a minor part) with regard to the process they make in order to arrive at that end result (e.g., their communication and reporting skills, problem-solving attitude, etcetera during the project). This "*process evaluation*" is built around a set of specific evaluation criteria for students of this faculty, based upon the pedagogic vision the faculty. More specifically, depending on the trajectory a student is following, the thesis advisor(s) need(s) to assess a student two or three times on 4 skill dimensions (each comprising of a set of specific skills to be rated from insufficient until very good) during the completion of his or her master thesis.

In this context, the *procesEval* application, based on NST, was created around 2013. Up to that moment, the process evaluation was either performed on paper or had to be registered via a customized part of the university's online learning and course management system. While the paper based evaluation was considered as generating administrative overhead (the results had to be manually copied into the university's database systems by the administration) and providing little overview for the thesis advisors (e.g., when performing the second process evaluation they could not easily consult the first process evaluation in order to make a more objective comparison), the electronic variant in the online learning and course management system was considered cumbersome from a usability perspective (e.g., users complaining about the amount of clicks required to perform "simple" actions or experiencing difficulties in order to find the information they are looking for).

The faculty management decided to develop an NST application to manage the process evaluations. This choice was made for several reasons. First, the expertise on how to build NST applications was present within the faculty itself as the theory (and the adjoining code expanders) was the output of research projects of faculty members. Second, as the software system would be developed by members of the faculty itself as well, the developers were highly knowledgeable about the inner working of the faculty (administration) and the associated (functional) requirements. And third, evolvability and maintainability were considered to be import quality aspects of the software system to be developed as the process evaluation was anticipated to remain an important part of the student evaluations for several years to come (but could be subject to some further fine-tuning or redirection in the future). Given the situation of the project as sketched above, it was expected that the application could be developed in a rather short development trajectory without too many hurdles (i.e., no significant risk related to the technology was present and the application domain was well known and understood).

The application itself was developed in the beginning of 2013. In the academic years 2013–2014 and 2014–2015, a first pilot test with a set of key users (technological proactive faculty members) was conducted. In the academic years 2015–2016 and 2016–2017, the set of test users was gradually expanded up to the level were all thesis supervisors could use the procesEval application if they wanted, but could still use the paper version if preferred. As of the academic year 2017–2018, all faculty members were expected to use the NST procesEval application for the administration of the master thesis process evaluations. Apart from minor (usability) adjustments, the project has been completed without major problems. Currently, on a yearly basis, about 45 faculty members manage the process evaluation of roughly 500 students via the procesEval application.

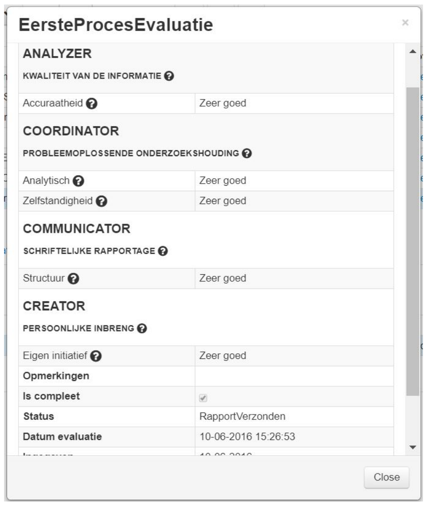
In Figure 1, a screenshot of the procesEval application is shown (the names of the students are blurred out to assure anonymity, the names of the label being Dutch as this is the administrative language of the organization). Here, one can see that a supervisor can get an overview of all the students he or she is supervising in the current academic year. By selecting a particular student, a set of tabs appears below the first table providing further details regarding his (earlier) evaluations or working sessions (e.g., meetings) and documents (e.g., preliminary thesis version). Figure 2 shows a screenshot of one particular process evaluation. The application therefore manages all process evaluations (typically 2-3) of all master dissertations (as of 2017–2018) of multiple academic



Naam student	Opleiding	Titel scriptie	Academiejaar	Promotor	Co promotor	Status	Duo thesis partner
[redacted]	HIB	Beheer van elektronische communicatie (e-mail) in de context van de Vlaamse Overheid	2014-2015	jan.verelst	philip.huysmans	Tweede procesevaluatie voltooid	
[redacted]	HIB	Datamining mogelijkheden voor Smart city - Stad Antwerpen	2014-2015	jan.verelst		Tweede procesevaluatie voltooid	
[redacted]	HIB	Een analyse van de succesfactoren van IT projecten en hoe IT project failure vermijden	2014-2015	jan.verelst	philip.huysmans	Tweede procesevaluatie voltooid	
[redacted]	HIB	Granulariteit	2014-2015	jan.verelst		Voor eerste procesevaluatie	
[redacted]	HIB	Het gebruik van de DEMO-methodologie in Enterprise Architectuur projecten	2014-2015	jan.verelst		Voor eerste procesevaluatie	
[redacted]	HIB	Onderzoek naar de implementatie van DEMO-gebaseerde Enterprise Architectuur a.d.h.v. een praktijkstudie	2014-2015	jan.verelst	philip.huysmans	Tweede procesevaluatie voltooid	
[redacted]	HIB	Bedrijfsprocessen personeelsplanning in de revalidatieinstelling	2015-2016	jan.verelst		Totale procesevaluatie voltooid	

Accuraatheid	Analytisch	Zelfstandigheid	Structuur	Eigen initiatief	Is compleet	Status	Datum evaluatie	Document
Zeer goed	Zeer goed	Zeer goed	Zeer goed	Zeer goed	✓	RapportVerzonden	10-06-2016 15:26:53	

Figure 1. A general screenshot of the procesEval application.



ANALYZER

KWALITEIT VAN DE INFORMATIE

Accuraatheid	Zeer goed

COORDINATOR

PROBLEEMOPLOSSENDE ONDERZOEKSHOUDING

Analytisch	Zeer goed
Zelfstandigheid	Zeer goed

COMMUNICATOR

SCHRIFTELIJKE RAPPORTAGE

Structuur	Zeer goed

CREATOR

PERSOONLIJKE INBRENG

Eigen initiatief	Zeer goed
Opmerkingen	
Is compleet	✓
Status	RapportVerzonden
Datum evaluatie	10-06-2016 15:26:53

Figure 2. A screenshot of a specific process evaluation within the procesEval application.

years. Based on the provided information, the application automatically generates overview reports of the evaluations, sends an email with information regarding their evaluations to students and supervisors as well as reminders (e.g., when a particular process evaluation is due).

IV. CASE ANALYSIS

A. General overview

An NST application typically consists of a set of base components (which are reused in several applications), as well as one or multiple non-base components (typically specific for the application under consideration). The base components used within the procesEval application consisted of 29 data elements, 7 task elements and 1 flow element. The non-base component used within the procesEval application consisted of 14 data elements, 8 task elements and 4 flow elements. As a consequence, relatively speaking, the NST application was still rather small comprising about 63 NST elements.

B. Model variations

By using the NST approach, the procesEval application could be extended and adapted at the level of the model (i.e., the definition of the different element instances for the considered application domain). For instance, additional elements could be added: next to the registration of three possible process evaluations for each student, some working documents and information regarding working sessions (e.g., what was agreed upon by the student and his supervisor during a meeting) could be added to the model. After re-generating the application based on this model, this functionality becomes available in the new version of the application. Similarly, existing (i.e., earlier created) components could be added to the model. For example, a notification component was added to the procesEval as this contained the functionality to automatically trigger emails and could be leveraged to enable the automatic report delivery (of the process evaluations to the students, supervisors and administration). While the model could be changed in terms of data elements and components, this also holds for all other possible changes within the model.

More specifically, the following types of adaptations can be performed to create different variations of the application:

- addition, update or deletion of a component (i.e., a set of data, task and flow elements);
- addition, update or deletion of a data element definition (its fields with its types and options, finders, options, child elements);
- addition or deletion of a task element definition (the specific implementation of a task is a crafting, see below);
- addition update or deletion of a flow element definition and its accompanying default state transitions.

It should be remarked that the determination and evolutions of such model is completely technology-agnostic (i.e., does not require any specification in programming language specific terminology). For instance, the specification of the model (in terms of elements and their properties) is currently stored in an XML format, not containing any references to the (background) technology of the current reference implementation (i.e., Java). Based on this model, boiler plate source code for each of the layers can be created.

C. Crafting variations

Once the model is converted into boiler plate source code, additional code (so-called "craftings", which are custom made for an application) could be added between predefined anchors (insertions) or in additional classes (extensions). This way, non-standard functionality can be incorporated within the application as well. In total, the procesEval contained 22 classes with insertions and 29 additional classes. For instance, in the procesEval application, specific coding had to be added to make sure that a supervisor logging into the application can only view those master dissertations that he is supporting in the concerning year (i.e., dissertations supported by other supervisors or those of the previous year should not be visible). For this purpose, a few lines of code were added in the MasterThesisFinderBean determining the fetching of the results viewable for a particular user. These FinderBeans are expanded as part of the data layer: enforcing the filter of master dissertations at the level of the data layer ensures that no data from other users can be retrieved by the currently logged in user. Consequently, this crafting only impacts the data layer, while the remaining layers have no impact from this change: they perform their functionality handling the (filtered) data offered by the MasterThesisFinderBean.

Additionally, a set of screentips was added to assist the user in filling-in the process evaluation (e.g., summarizing the meaning of each of the evaluation criteria in case of a mouse-over). The expanded NST code base supports this functionality by providing a helpInfo Knockout binding. Specific screentips can be added by including a crafting using this Knockout binding, and referring to a certain key. At run-time, the specific values for the required keys can be added in instances of HelpInfo data elements. This enables the configuration of the screentips even when the application has already been deployed. Note that only the view layer is customized for this functionality. This makes sense, since it is purely a useability concern, not impacting actual business logic. However, it is dependent on the specific technology used in the view layer (i.e., Knockout), and should be remade when a different technology is used.

Next, as mentioned before, the procesEval also needed to create and send reports summarizing the content of the process evaluation. The definition of these reports (the items to be included and the corresponding layout) is considered to be a separate functionality, and should therefore be contained in a task element. The expanders provide all boilerplate code needed to execute this task in the NST application, and only the specific report generating functionality needs to be added as a crafting. The actual implementation of the execution of a task element is clearly separated, allowing versions and variations of the task implementation to co-exist. Currently, reports are generated using Jasper Reports. This requires the addition of a Jasper template file to the code base, and some code to fill the parameters to be inserted into this template. The additional processing logic is completely contained in the logic layer.

These craftings were added in a gradual and iterative way to the application: each time a particular additional functionality was added or improved, a new version of the overall application could be built and deployed. Each of these craftings were situated at another layer (i.e., data, view and logic).

D. Infrastructural technology variations

The procesEval application could be generated by using various different underlying infrastructural technologies. For instance, whereas a prototype of the application is typically demonstrated by using an HSQL database, most production systems are deployed while using a PostgreSQL database. Nevertheless, one can choose for SQLServer and MySQL databases as well. Further, the procesEval can be built by using different build automation frameworks (i.e., Ant and Maven). And finally, the procesEval could also be generated by using different controlling (Cocoon, Struts2, or combination Struts2-Knockout) and styling frameworks (plain style or using Bootstrap). In practice, the Struts2-Knockout and Bootstrap were used in the production environment. Changing the choice of a particular infrastructural technology in the procesEval only impacts those layers depending on the purpose of the technology (e.g., the database selection impact the data layer, whereas the GUI framework selection impacts the view layer).

E. Expander version variations

The expanders (i.e., the programming logic used to convert the model into boiler plate source code according to the infrastructural technologies chosen) evolves throughout time as well. This way, when considered the current procesEval project duration (2013–present), 8 different production versions were deployed while using the same model and craftings (as the expanders provide backwards version compatibility). In each of these production versions, the new or improved possibilities of the expanders could be used. For instance, in one particular version of the expanders, information regarding a Date field did no longer have to be entered manually but could be selected by using a more advanced date picker. And, more relevant in the context of the procesEval, another particular version of the expanders allowed the automatic creation of summarizing graphs on certain fields. For example, it would now be possible to inspect the number of master dissertations who did not yet receive a first process evaluation versus those who did in a visual way. In order to use the date picker, no changes in the model or the craftings are required. In order to use the status graphs, only one additional specification in the model (i.e., an option indicating that a graph for a particular field should be created) needs to be added. Clearly, the precise set of layers that is impacted due to an expander update depends on the type of modifications performed in that particular version update (logic related, view related, etcetera).

V. DISCUSSION

While the above analysis is only based on one case study, we anticipate that the proposed categorization can be generalized to a large extend as it also aligns with the general "degrees of freedom" available during the development and maintenance of an NST application. This overall approach, together with 4 variability dimensions, is visualized in Figure 3.

First, as represented at the top of the figure, the modeler should select the *model* he or she wants to expand. Such a model is technology agnostic (i.e., defined without without any reference to a particular technology that should be used) and represented by a blue puzzle (i.e., each puzzle piece represents a defined element, with the columns corresponding to data, task, flow, trigger and connector elements). Such a model can have multiple versions throughout time (e.g., being updated or complemented) or concurrently (e.g., choosing between a more extensive or summarized version). As a consequence, the figure contains multiple blue puzzles that are put behind each other and the chosen model represents a variability dimension (represented by the green bidirectional arrow).

Second, the *expanders* (represented by the trapezoid in the figure) generate (boiler plate) source code by taking the specifications in the chosen model as its *arguments*. For instance, for a data element Person, a set of java classes PersonBean, PersonLocal, PersonRemote, PersonDetails, etcetera will be generated. This code can be called boiler plate code as it provides a set of standard functionalities for each of the elements within the model. Nevertheless, one could argue that this set of standard functionalities is already quite decent as it contains the possibilities to provide standard finders, master-detail (waterfall) screens, certain display options, document upload/download functionality, child relations, etcetera. The expanders themselves evolve throughout time. Typically, in each new version, a set of bugs of the previous one are solved and additional features (e.g., creation of a status graph) are provided. It should be remarked that, given the fact that the application model is completely technology agnostic and can be used as argument for any version of the expanders, these bug fixes and additional features become available for all versions of all application models (only a re-expansion or "rejuvenation" is required). As a consequence, the figure contains multiple trapezoids that are put behind each other and the expander version represents a variability dimension (represented by the green bidirectional arrow).

Third, in the middle left of the figure, a set of *infrastructural options* are displayed by means of different rectangular blocks. These consist of global options (e.g., determining the build automation framework), presentation settings (determining the graphical user framework), business logic settings (determining the database used) and technical infrastructure (e.g., determining the background technology). For each of these infrastructural options, the modeler can choose out of a set of possibilities (e.g., different user interface frameworks for which the associated code can be generated), which will be used by the expanders as their *parameters*. That is, given a chosen application model version and expander version, different variants of boiler plate code can be generated, depending on the choices regarding the infrastructural options. As a consequence, the figure contains multiple infrastructural option block sets that are put behind each other and the infrastructural options represent a variability dimension (represented by the green bidirectional arrow).

Fourth, *craftings* ("custom code") can be applied to the generated source code. These craftings are represented in the lower left of the figure by means of red clouds as they enrich (are put upon) the earlier generated boiler plate code and can be harvested into a separate repository before regenerating the software application (after which they can again be applied). This includes extensions (e.g., additional classes added to the generated code base) as well as insertions (i.e., additional lines of code added between the foreseen anchors within the code). Craftings can have multiple versions throughout time (e.g., being updated or complemented) or concurrently (e.g., choosing between a more advanced or simplified version). These craftings should contain as little technological specific statements within their source code (apart from the chosen background technology). Indeed, craftings referring to (for instance) a specific GUI framework will only be reusable as long as this particular GUI framework is selected during the generation of the application. In contrast, craftings performing certain validations but not containing any EJB specific statements will be able to be reused when applying other versions or choices regarding such framework. Craftings not dependent on the technology framework of a specific layer can be included in the "common" directory structure, whereas technology-dependent craftings need to reside in the directory structure specified for that technology (e.g., EJB for the logic layer, JPA for the data layer, Struts2 for the control layer). As a consequence, the figure contains multiple crafting planes that are put behind each other and the chosen set of craftings represents a variability dimension (represented by the green bidirectional arrow).

In summary, each part in Figure 3 with green bidirectional arrows is a variability dimension in an NST context. It is clear that talking about *the* "version" of an NST application (as is traditionally for software systems) in such context becomes rather pointless. Indeed, the eventual software application (the grey puzzle at the bottom of the figure) is the result of a specific version of an application model, expander version, infrastructural options and set of craftings. Put differently, with M, E, I and C referring to the number of available application model versions, the number of expander versions, the number of infrastructural option combinations and crafting sets respectively, the total set of possible versions V of a particular NST application becomes equal to:

$$V = M \times E \times I \times C$$

Remark that the number of infrastructural option combinations is equally a product:

$$I = G \times P \times B \times T$$

Where G represents the number of available global option settings, P the number of presentation settings, B the number of business logic settings and T the number of technical infrastructure settings. This general idea in terms of combinatorics corresponds to the overall goal of NST: enabling evolvability

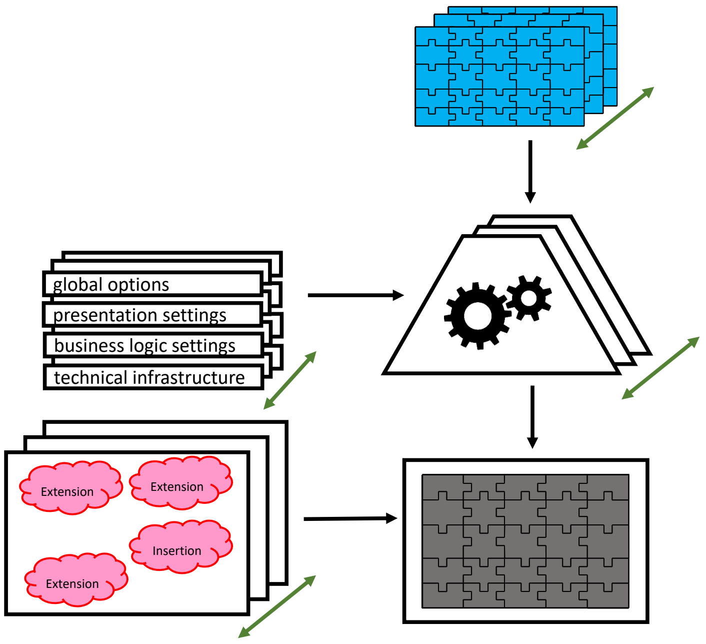


Figure 3. A graphical representation of four variability dimensions within a Normalized Systems application.

and variability by *leveraging the law of exponential variation gains* by means of the thorough decoupling of concerns and the facilitation of their recombination potential [4].

VI. Conclusion

This paper presented a case study of an NST software application in an educational context and analyzed the different dimensions in which it could evolve. Based on this, 4 general variability dimensions were proposed.

This paper is believed to make several contributions. From a theoretical side, inductive reasoning based on our case allowed the formulation and illustration of 4 variability dimensions, might be the (or at least a subset of the) orthogonal dimensions along which a typical NST application can evolve. At the same time, these variability dimensions clarifies that the concept of an overall application "version" is not applicable for NST applications as a specifically deployed application is the result of a combination of choices for each of the variability dimensions. For practitioners, this paper contributes to the set of case studies available on NST, which might provide them with a better insight regarding the application potential of the theory in practice.

Next to these contributions, it is clear that this paper is also subject to a set of limitations. That is, we proposed the set of variability dimensions based on one case study, which was limited in size and complexity. This limits the generalizability of our findings. Therefore, future research should be directed towards the analysis of additional cases, including information systems being larger, more complex and executed within other application areas than the educational industry. These additional cases might confirm, and possibly extend, the variability dimensions proposed in this paper.

References

[1] R. Agarwal and A. Tiwana, "Editorialevolvable systems: Through the looking glass of is," Information Systems Research, vol. 26, no. 3, 2015, pp. 473–479.

[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. 76, no. 12, 2011, pp. 1210–1222, special Issue on Software Evolution, Adaptability and Variability.

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

[4] H. Mannaert, J. Verelst, and P. De Bruyn, Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design. Koppa, 2016.

[5] M. Op't Land, M. Krouwel, E. Van Dipten, and J. Verelst, "Exploring normalized systems potential for dutch mods agility: A proof of concept on flexibility, time-to-market, productivity and quality," in Proceedings of the 3rd Practice-driven Research on Enterprise Transformation (PRET) working conference, Luxemburg, Luxemburg, September 2011, pp. 110–121.

[6] G. Oorts, P. Huysmans, P. De Bruyn, H. Mannaert, J. Verelst, and A. Oost, "Building evolvable software using normalized systems theory : a case study," in Proceedings of the 47th annual Hawaii international conference on system sciences (HICSS), Waikoloa, Hawaii, USA, 2014, pp. 4760–4769.

[7] P. Huysmans, P. De Bruyn, G. Oorts, J. Verelst, D. van der Linden, and H. Mannaert, "Analyzing the evolvability of modular structures : a longitudinal normalized systems case study," in Proceedings of the Tenth International Conference on Software Engineering Advances (ICSEA), Barcelona, Spain, November 2015, pp. 319–325.

[8] P. Huysmans, J. Verelst, H. Mannaert, and A. Oost, "Integrating information systems using normalized systems theory : four case studies," in Proceedings of the 17th IEEE Conference on Business Informatics (CBI), Lisbon, Portugal, July 2015, pp. 173–180.

[9] P. De Bruyn, P. Huysmans, and J. Verelst, "Tailoring an analysis approach for developing evolvable software systems : experiences from three case studies," in Proceedings of the 18th IEEE Conference on Business Informatics (CBI), Paris, France, August-September 2016, pp. 208–217.

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