Systematic Rejuvenation of a Budgeting Application over 10 years: A Case Study

Chetak Kandaswamy, Jan Verelst

Department of Management Information Systems
Faculty of Business and Economics
University of Antwerp, Belgium
Email: jan.verelst@uantwerpen.be

***Abstract*—Normalized Systems (NS) theory has recently been proposed as a means of increasing software agility. NS theory posits that software evolvability, or the ease with which software can be changed, can be achieved by adhering to a set of theorems that result in a specific and evolvable software architecture, based on the use of NS-specific code generators called expanders. While the theoretical contributions of NS theory have been well-documented in previous research, there are few reports on real-life cases where NS theory has been employed. This paper documents a development project that demonstrates the feasibility of the NS approach for building evolvable software and highlights the benefits of a real-life NS development project over a period of more than 10 years, in which the system was built and afterwards regenerated using the NS code generators. The results confirm the feasibility of systematically regenerating information systems in Java over time with limited resources, eliminating or drastically reducing the need for rebuilds from scratch, in order to deal with structure degradation of information systems, more specifically for information systems of limited size and complexity, which are commonplace in today's digital economy.**

Keywords-Normalized Systems; Evolvability; Agility; Software Rejuvenation

I. INTRODUCTION

In recent years, there has been a growing body of research on agile software development. While this research has yielded valuable insights into improving agile development processes, there has been comparatively less focus on enhancing the agility of the software itself. Important Agile frameworks, such as the Scaled Agile Framework (SAFe), define Agile Architecture as a set of values and principles that support the active evolution of the design and architecture of a system while implementing new capabilities. This definition points more in the direction of a process than it does in assuring that the system itself will be agile. In that respect, Agile Architecting is a better term to refer to an agile way of doing architecture, and Agile Architecture could point to the intentionality of creating an evolving system.

Normalized Systems (NS) theory has recently been proposed as a means of increasing software agility. NS posits that software evolvability, or the ease with which software can be changed, can be achieved by adhering to a set of theorems that result in a specific and evolvable software architecture. This architecture offers systems theoretical stable responses to changing business and/or technical requirements. NS theory has been refined and extended over the years and has been implemented in several software projects. While the theoretical contributions of NS theory have been well-documented in previous research, there are few reports on real-life cases where NS theory has been employed.

This paper documents a development project that demonstrates the feasibility of the NS approach for building a Budgeting application and maintaining it over a period of 10 years.

The paper is structured as follows. In Section II, we review the concepts behind NS theory and software rejuvenation. Section III will provide information about the Budgeting application and an overview of the different changes applied to the application over a period of 10 years. In Section IV, we will discuss the Budgeting application from the perspective of the owner of the application, the Province of Antwerp, and report their reflections on the past 10 years. Section V presents our conclusions.

II. FUNDAMENTALS OF NS THEORY

Software architectures should be able to evolve as business and technical requirements change over time. In NS theory, evolvability is measured by a lack of Combinatorial Effects (CE) in software architectures. Combinatorial Effects constitute a specific kind of ripple effects: when the impact of a change, measured in the number of impacted modules, depends not only on the type of change but also on the size of the software system, a Combinatorial Effect occurs. NS theory assumes that software undergoes unlimited evolution (i.e., that both new and changed requirements will make a software system increase in size over time), which makes Combinatorial Effects very harmful to software evolvability. Indeed, if changes to a system depend on the size of the growing system, these changes become harder to handle (i.e., requiring more work and therefore lowering the evolvability of the system).

NS theory is built on principles from systems theory (stability) and statistical thermodynamics (entropy). In systems theory, a system is stable if it has bounded input leading to bounded output (BIBO). NS theory applies this idea to software design as a bounded change in functionality should only cause a bounded change in the software. In systems theory, stability is measured at infinity. NS theory considers systems that grow infinitely large over time and will go through infinitely many changes. According to NS theory, a system is stable towards changes, if it does not have CE, meaning that the effect of a change only depends on the type of change and not on the system size.

NS theory suggests four theorems and five elements as the basis for creating evolvable software through pattern expansion of the elements. The theorems have been proven formally, and provide a set of design guidelines that must be followed strictly in order to avoid Combinatorial Effects. The NS elements offer a set of predefined higher-level structures, patterns, or "building blocks", that provide functionality while conforming to all NS theorems. Therefore, they constitute a blueprint for implementing the core functionalities of realistic information systems.

A. *NS Theorems*

NS theory proposes four theorems that describe the necessary conditions for software to be free of Combinatorial Effects:

- Separation of Concerns
- Data Version Transparency
- Action Version Transparency
- Separation of States

Violation of any of these 4 theorems will lead to Combinatorial Effects and thus less evolvable software under change.

B. *NS Elements*

Consistently adhering to the four NS theorems seems very challenging for developers because of several reasons. First, following the NS theorems leads to a fine-grained software structure as concerns and states are separated, which does introduce some development overhead that may slow down the development process. Second, the theorems must be followed all the time, which is problematic in a context where human programmers work under varying project conditions, including (occasionally) limited time and budgets. Third, the accidental introduction of Combinatorial Effects results in an exponential increase of rework that needs to be done at a later time.

Five elements were therefore proposed which make the realization of NS applications more feasible, as they can be instantiated by code generators called expanders. These elements are carefully engineered patterns that comply with the four NS theorems and that can be used as essential building blocks for a wide variety of applications. The elements are named according to the elementary functionality they offer: data element, action element, workflow element, connector element, and trigger element.

- **Data Element**: the structured composition of software constructs to encapsulate data into a module (including get- and set methods, persistency, exhibiting version transparency,etc.).
- **Action Element**: the structured composition of software constructs to encapsulate an action into a module.
- **Workflow Element**: the structured composition of software constructs describing the sequence in which a set of action elements should be performed to fulfill a flow, into a module.
- **Connector Element**: the structured composition of software constructs into a module allowing external systems to interact with the NS system without calling components in a stateless way.
- **Trigger Element**: the structured composition of software constructs into a module that controls the states of the system and checks whether any action element should be triggered accordingly, e.g., based on time conditions.

The element not only provides core functionality (such as persistency of data, execution of an action, etc.) but also addresses the cross-cutting concerns that each of these core functionalities require to function properly. As cross-cutting concerns cut through every element, they require careful separation from other concerns in order not to introduce Combinatorial Effects.

C. *Element Expansion*

An application is mainly composed of a set of data, action, workflow, connector, and trigger elements that realize its requirements. An NS expander instantiates the software elements into source code for the specific application. The expanded code will provide functionalities specified in the application definition and constitutes a fine-grained modular structure that follows the NS theorems (see Figure 1) and is therefore free from combinatorial effects. This generated part of an application is also called the skeleton of the application.

Next, remaining functionality, such as the business logic for the application, is manually programmed or customized inside the expanded modules, at pre-defined locations. This functionality is called a customization or crafting. The presence of combinatorial effects in this manually programmed part of the application, depends on the adherence of the individual programmer to the NS theorems. However, a strength of this approach is that the only location where Combinatorial Effects can be introduced, is in the customized code.

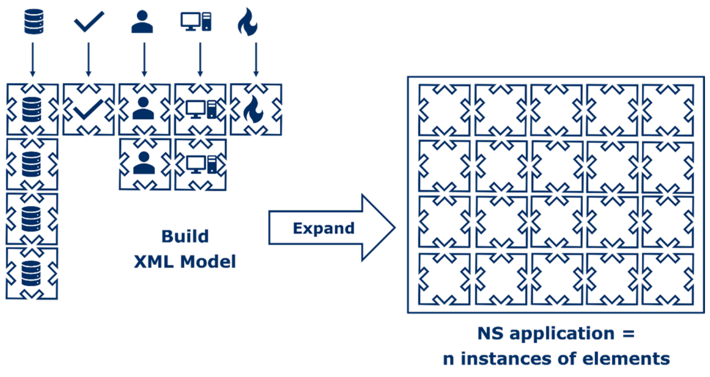

Figure 1. Requirements expressed in an XML description file, used as input for element expansion

D. *Harvesting and Software Rejuvenation*

The expanded skeleton has some pre-defined places where customizations can be made. To keep these customizations from being lost when the application is re-expanded at a later time, these customizations are gathered and put back when the application is re-expanded. This process of gathering and putting back the customizations is called harvesting and injection.

The application can be re-expanded for different reasons. For example, the code templates of the elements are improved (bug fixes, performance improvements, new versions of supporting technologies, or changes in the technology, such as a new persistence framework, etc.).

The purpose of software rejuvenation is to carry out the harvesting and injection process routinely to ensure that the improvements of the 5 element code templates are incorporated into the skeleton of the application.

In our experience, in a Java environment, expansion produces more than 80% of the code of a production-ready application. The expanded code can be called boiler-plate-code, but it is more complex than what is usually meant by that term because it deals with cross-cutting concerns such as persistency, remote access, logging and security at an advanced level. The manual production of such code often is time consuming. Using NS expansion, this time can now be spent on, e.g., the constant improvement of the element code templates, the development of new code templates that make the elements compatible with new technologies, and on meticulous coding of the business logic. The changes in the elements can be applied to all expanded applications, giving the concept of code reuse a new meaning. A modification on a code template by one developer can be used by all developers on all their applications, with minimal impact, thanks to the rejuvenation process. Figure 2 summarizes the NS development process.

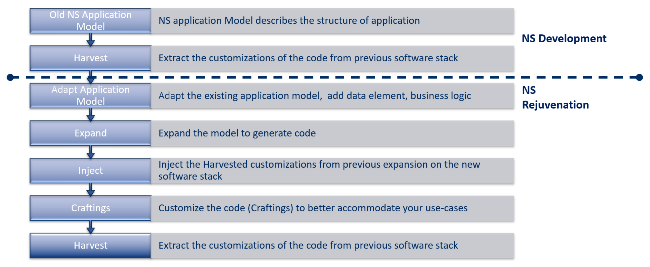
Old NS Application Model
NS application Model describes the structure of application
NS Development
Harvest
Extract the customizations of the code from previous software stack
Adapt Application Model
Adapt the existing application model, add data element, business logic
NS Rejuvenation
Expand
Expand the model to generate code
Inject
Inject the Harvested customizations from previous expansion on the new software stack
Craftings
Customize the code (Craftings) to better accommodate your use-cases
Harvest
Extract the customizations of the code from previous software stack

Figure 2. The NS development process

III. THE USE CASE: PROVINCE OF ANTWERP BUDGETING APPLICATION

In this section, we first describe the Budgeting Application at a functional level, and then describe the evolution and rejuvenation process that took place over the course of about 10 years. This case study is based on interviews with the Head of IT Projects of the Province of Antwerp as well as the programmers who were involved in development and maintenance.

A. The Application

As a case study for software rejuvenation, we selected a Budgeting application of a local Belgian government. The application was built because the existing financial ERP package was difficult to adapt to the specifics of Belgian government budgeting regulations. The application was first built using NS technology in 2012 and is currently still in use. The functional requirements of the application are budget creation and management, expense tracking and control, managing different revenue streams, forecasting and planning, reporting and analysis, compliance and audit trail, integration with financial systems, data security, and privacy.

This Budgeting application has played a crucial role in enhancing transparency, accountability, and efficiency in the budgeting process. It has enabled the government to monitor and manage its financial resources effectively, ensure compliance with fiscal policies, and make data-driven decisions to allocate resources efficiently. This application has integrated well with the existing financial systems used by government entities, such as accounting software or Enterprise Resource Planning (ERP) systems. This integration has ensured data consistency and has reduced manual data entry.

The functional requirements are easily explained using the Entity Relationship Diagram (ERD) shown in Figure 3. The diagram shows the Budget as the central data element instance of the application. The Budget element is defined by a combination of the following 11 data elements: Article, Budget type, Budget change, Budget year, Cell, Domain, Product, Recording, Service, Supplier, and Team. The unique combination of these 11 parameters is the key to the budget in its most basic manifestation.

The current budget is an aggregation of many sub-budgets over time along with the combination of the above parameters in real-time for data integrity reasons. The calculated current budget is not stored in the database to avoid error propagation which may lead to faulty data. The most granular budget is calculated based on the following data elements: Article, Budget type, Budget change, Budget year, Department, Domain, and Product instances as visualized on the left of the figure. The specific budget belongs to a single department, activity, etc. The activity is grouped with the Economic groups, which in turn makes the Budget estimate.

B. Application evolution and rejuvenation

Over the past 10 years, the Budgeting application has been subject to many changes. Although the business logic of the application, mainly driven by legislation, required only one major update over this period, the number of changes in user functionalities were more frequent. Both changes in legislation and user functionalities required new code customizations. Also, there have been many changes to the element code templates. They have been updated based on feedback from customers (bug reports, performance issues, etc.) and the changing technological landscape (new operating system versions, database updates, programming language evolutions, application server changes and even the switch in deployment methods from onsite to cloud).

Figure 4 summarizes the software rejuvenation of the Budgeting application over the past 10 years. The efforts

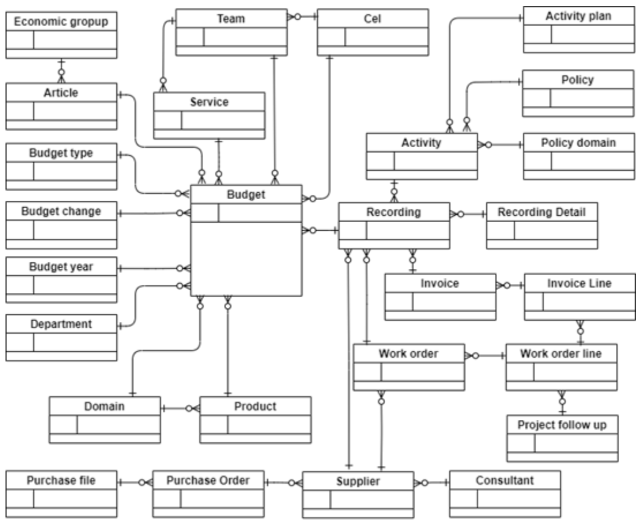

Figure 3. ERD model of the Budgeting application

required to perform technology updates using rejuvenation add up to a total of 5 days over 10 years. Between 2012 and 2019, the technologies used by the application have endured, which contributes to the relatively limited effort required. The rejuvenation mainly included updates of the element code templates, benefiting from the continuous evolution of element code template improvements done by other developers. In 2019, a significant update at the technology side happened: a change in the programming language version, application server, and frontend technology. The total effort was 2 man-days to accomodate these changes. In 2023, the changes in technology were even more profound as the programming language version, frontend, database, application server, and deployment method (container instead of server-based) changed. The effort was only 1 man-day. In summary, the skeleton of the Budgeting application was rejuvenated several times over the past 10 years, each time requiring an effort in terms of one to several man-days, which can be considered a limited investment to incorporate all the benefits of a rejuvenation described above.

The total time invested in changes to customizations or craftings adds up to 50 man-days from application conception (2012) to the current state (2023). The effort of implementing new customizations (new legislation in 2014) and user functionalities (2014, 2015 and 2019) can be considered similar to whatever development method and/or technology was used in the industry at that time, which is unsurprising as this essentially manually written code in an NS application.

In summary, over a period of 10 years, the total effort of change has been 28 man-days, of which 23 have been purely functional changes and 5 due to rejuvenation. These figures confirm and even outperform estimations that were made about the development effort of this very same application in 2014 (see Figure 5) [4].

IV. VOICE OF THE CUSTOMER

This section is based on interviews with the Head of IT Projects and Solutions at the Province of Antwerp.

1) On the advantages of Rejuvenation using NS framework: “The main advantage for us was the speed that can be gained with the rejuvenation of the application. Because the process of expansion and re-injection is fully automated and fast, a new version can be put in place and the actual functionality can be tested instead of also having to validate and test the boiler-plate code.”

2) On developing with or without NS: “We have no real data concerning the effective difference between development with or without NS. In my opinion, if we did not use NS, the first change of the application in 2014 (new budgeting legislation), would have resulted in building a new application, instead of just rejuvenating the existing one. Such a rebuild would have probably taken 50 man-days. While with rejuvenation, we only had a few days of functional testing to do.”.

3) On Maintenance cost: The maintenance of 6 different applications at the Province of Antwerp built using the NS methods (including the Budgeting application) required only 4 man-days of maintenance operations both in 2021 and in 2022 (across all 6 applications).

4) On NS vs. Low Code: As the proprietary budgeting tool was to be used by only 20 users, low-code and no-code platforms could also have been considered as development platforms, as they allow users to create applications using a minimal amount of coding. At the time of development (2012), such platforms were not considered by the Province of Antwerp. Revisiting the NS vs. low-code decision at this point in time, can be done based on a number of criteria. First, it is important to note that stakeholders from the Province of Antwerp required specific customizations for the Budgeting application, potentially causing low-code platforms to be challenged in terms of customization, scalability, and flexibility. Second, if an organization builds applications heavily reliant on the low-code platform’s proprietary features or architecture, migrating to a different platform or transitioning away from the platform can be difficult and time-consuming.

5) On NS vs. Shadow IT: The Budgeting application is not a challenging and complex application, and one might be tempted to turn to the usage of a MS Excel or MS Access-based application, completely created and maintained by the business, instead of IT. The Province of Antwerp did not go down this path as they already had some years of experience in doing their budgeting work in MS Excel and noticed important drawbacks such as the fragility of the solution, dependence on a few people who master the implementation of the business logic in MS Excel and high maintenance cost.

V. CONCLUSION

In this paper, we discussed how the NS theory can be applied to rejuvenating a Budgeting application, which essentially is a small CRUDS application, which was built using the NS expanders. Over this period, this application has undergone multiple functional and non-functional, technical

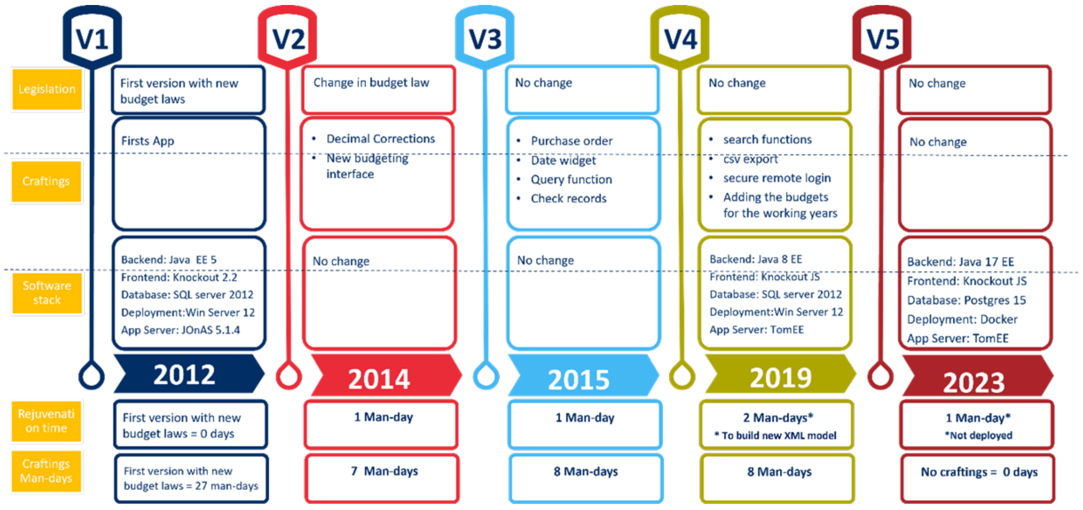

Figure 4. Summary of Software Rejuvenation for 10 Years

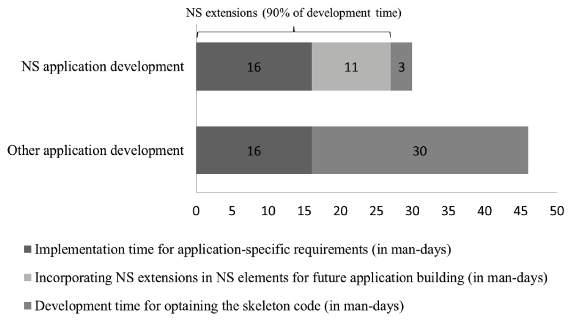

Figure 5. Comparison of estimated development time [4]

changes. The technical changes were limited in the sense that updates from technologies were required, but no major shifts to other technologies. Nonetheless, in a time where many applications are rebuilt after 5-10 years, it is interesting to see that it is feasible to see that rejuvenation is feasible over a period of 10 years, with the skeleton of the application being updated to the most recent version of the underlying technologies. This suggests that the increased use of code generators holds significant promise for the future.

REFERENCES

[1] H. Mannaert, J. Verelst, and P. De Bruyn, "Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design", Koppa Publishing, ISBN 978-90-77160-09-1, 2016.
[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, Volume 76, Issue 12, pp. 1210-1222, 2011.
[3] P. Huysmans, G. Oorts, P. De Bruyn, H. Mannaert, and J. Verelst, "Positioning the normalized systems theory in a design theory framework", Lecture notes in business information processing, Springer, ISSN 1865-1348-142, pp. 43-63, 2013.
[4] G. Oorts, et al., "Building Evolvable Software Using Normalized Systems Theory: A Case Study", Proceedings of the annual Hawaii international conference on system sciences, ISBN 978-1-4799-2504-9, pp. 4760-4769, 2014.