On the Evolvability of Code Generation Patterns: The Case of the Normalized Systems Workflow Element

Herwig Mannaert and Peter De Bruyn

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

Koen De Cock and Tim Van Waes

Research and Development Department
Normalized Systems eXpanders
Antwerp Science Park, Belgium
Email: {koen.de.cock,tim.van.waes}
@nsx.normalizedsystems.org

***Abstract*—Normalized Systems Theory (NST) aims to create software systems exhibiting a proven degree of evolvability. The theory contains a set of formally proven theorems and proposes a set of elements (patterns) to realize the adherence to the theorems in practice. While the development and evolution of several information systems in practice (based on the theory) have been documented, the evolution or enhancement regarding one of the fundamental element patterns has not yet been presented. Therefore, this paper discusses the evolution of one of these patterns, the flow element, and what this means for the software applications it is used in. This way, additional insight is provided on how the theory enables the enhancement and simultaneous evolution of large sets of information systems.**

Keywords–Evolvability; Normalized Systems; Design Patterns

I. INTRODUCTION

Having evolvable information systems (IS) is important for the survival chances of organizations, although the topic has not yet received much attention within the IS research area [1]. Normalized Systems Theory (NST) precisely focuses on providing an ex-ante proven approach to build evolvable software systems by using concepts from systems theory and statistical thermodynamics [2]–[4]. In order to apply the theory in practice, a code generation framework (NS expanders) was developed, compliant with NST. During the last five years, about 50 information systems have been developed using this code generation framework, from which some cases have been documented in previous publications [5]–[9].

While these cases describe the evolution of various versions of specific information systems developed using NST and the NS expanders, no specific evolution within the code generation framework itself has currently been documented. Therefore, this paper will discuss the evolution (the incorporation of additional and improved functionality) of a specific part of the NS expanders, i.e., the flow element. This way, we aim to provide additional insight into how the theory (and more specifically, the systematic improvement of the NS expanders) enables the enhancement and simultaneous evolution of a large set of information systems.

The remainder of this paper is structured as follows. In Section II, we discuss some related work with an emphasis on NST, which is the theoretical basis of both the evolutionary approach and the code generation framework. Section III explains the structure of an initial version of the flow element. Next, we discuss some additional developments and improvements regarding the considered element in Section IV and some empirical test results in the context of this element evolution in Section V. Finally, our discussion is offered in Section VI.

II. RELATED WORK

In Sections II-A and II-B, we introduce and summarize the theoretical underpinnings of NST and the associated expansion and regeneration mechanisms, respectively. In Section II-C, we briefly discuss the contrast with some existing approaches.

A. Normalized Systems Theory

The software applications and the patterns they make use of, as we will discuss and analyze in the following section, are based on NST. NST was proposed with the purpose of allowing the design of software applications exhibiting ex-ante evolvability [2]–[4]. In particular, the theory focuses on the ripple effects (due to all kinds of coupling) occurring in software systems when changes are applied and proposes some ways to eliminate them. It is believed that such kind of ripple effects may be one of the main causes of Lehman's Law of increasing complexity [10], which states that software systems become more difficult to maintain and adapt over time due to its deteriorating structure. Indeed, the more coupling and the more ripple effects, the more difficult software applications can be adapted.

NST starts from the concept of stability as defined in systems theory as its theoretical basis. A system is considered to be stable when a bounded input only results in a bounded output, even for those cases where an unlimited time period is considered. In the context of software applications, this would require that a bounded set of elementary functional changes should only result in a bounded impact to the software system, even when considering an unlimited time period (and therefore, an unlimited system size with an unlimited amount of software construct instances). This reasoning, based on stability, therefore also implies that the impact of changes to a software system (i.e., the number of construct instances that need to be created or adapted) cannot be dependent on the size of that information system. The impact should only be dependent on the size and property of the changes that are applied and not the number of construct instances within the system. Changes of which the impact is dependent on the size of the software systems are called *combinatorial effect*,

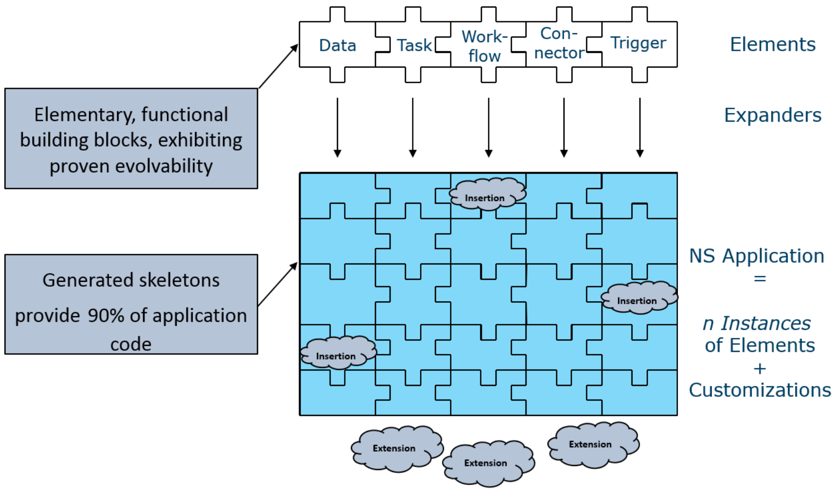
Data
Task
Work-flow
Con-nector
Trigger
Elements
Expanders
Elementary, functional building blocks, exhibiting proven evolvability
Generated skeletons provide 90% of application code
Insertion
Insertion
Insertion
NS Application = *n Instances* of Elements + Customizations
Extension
Extension
Extension

Figure 1. A graphical representation of code generation and additional custom coding within a Normalized Systems application.

and are to be avoided. NST has proposed and proved that a set of theorems should be complied with in order to avoid combinatorial effects (as their presence negatively impacts the evolvability of a software system) [2]–[4]:

- *Separation of Concerns*, which states that each concern (i.e., each change driver) needs to be encapsulated in an element, separated from other concerns;
- *Action Version Transparency*, which declares that an action entity should be updateable without impacting the action entities it is called by;
- *Data Version Transparency*, which indicates that a data entity should be updateable without impacting the action entities it is called by;
- *Separation of States*, which states that all actions in a workflow should be separated by state (and called in a stateful way).

B. NS Expansion and Regeneration

Applying the NST theorems in a systematic way results in a software system having many (but small) modules. The design of such fine-grained structure is far from trivial. Moreover, as any theorem violation during the development process results in combinatorial effects, it is very hard to create a perfectly NST compliant application by manual coding. Consequently, NST advocates the use of elements: design patterns which are re-used (and parametrized) over and over again to build a software system. Therefore, a software system is said to be generated or "expanded" to a large extent. In particular, the following set of five elements is currently proposed [3], [4]:

- *data element*: used to enter, update and retrieve data, such as *invoices* or *customers*.
- *task element*: used to perform specific tasks, such as creating and rendering an invoice.
- *flow element*: used to sequence and execute various tasks on the instances of the data.
- *connector element*: used to enable input/output by human users or external systems.
- *trigger element*: used to activate flows and tasks in a periodic way.

A code generation framework, called the NS expanders and built by a spin-off company of the University of Antwerp (i.e., the Normalized Systems eXpanders factory or NSX), has been created in order to further refine the elements. This framework allows the creation of NST software in a relatively straightforward way and has been used to generate several applications in different types of industries. As schematically represented in Figure 1, the generated code base or skeleton is later on augmented with additional —manually written— custom code. This custom code is divided into *insertions* (code fragments embedded within the classes of the elements) and *extensions* (separate software classes).

Insertions and extensions are harvested and stored in a separate source code repository, enabling the possibility to reinject this custom code base into future generations of skeletons consisting of expanded elements. Such future versions could provide additional or improved functionality in the element patterns (e.g., providing more advanced security features, or allow the use of new (versions of) frameworks for specific concerns like persistency or access control). This process of regenerating the code skeleton using a new version of the NS expanders, and reinjecting the existing custom code into a new version of the information system, is called *regeneration* or *rejuvenation*.

C. Related approaches

Other approaches than NST have obviously also already advocated the use of (software) design patters. Typically, the idea is to document and provide access to high quality solutions for frequently occurring problems so that the same problem does not need to be solved by every single developer and he or she can immediately make use of a mature solution that has proven its value in the past. A seminal work in this respect was for instance the work of the Gang of Four [11]. While these design patterns represent the core reasoning of the solution to a particular problem, most of them (such as those from Gamma et al. [11]) still require a certain amount of interpretation before they can be converted into actual working code (i.e., they cannot be mapped one-to-one to software code). The NS expanders, however, do not need this additional interpretation and result directly in operational software code.

Recently, some work on Model-Driven Software Development (MDSD) [12] has adopted a similar approach in which models are created and can then be converted into working code. Our approach differs in the sense that we specifically focus on generating software with a high degree of evolvability.

III. Initial Flow Element Pattern

This section focuses on the initial design of the *flow element* pattern and the additional requirements which arose during its use in practice. In the next section, we will discuss how these additional requirements were incorporated in an improved version of the element.

A. The Flow Element Pattern

Figure 2 presents a sequence diagram documenting an initial version of the inner pattern of the flow element (i.e., before the target development as discussed in Section IV was applied). For every flow element called `<Flow>`, a number of Java classes are generated or *expanded*, to implement the automated processing of a state machine operating on a data element called `<Data>`. The various operations or tasks of the state machine are specified in the various entries of a configuration data element called `StateTask`. The individual operations or tasks are implemented in task elements called `<Task>`, and every execution of a task on an instance of the data element `<Data>` is logged in an entry of an history data element called `<Data>TaskStatus`. In order to be able to start/stop the processing of the flow element and set some other parameters (such as time windows or time intervals), a control data element `EngineService` is provided.

Let us focus on the `orchestrate` method of the central class `<Flow>EngineBean` of the flow element `<Flow>` as shown in Figure 2. First, based on the name of the flow, the control data is retrieved from the appropriate entry of the *EngineService* data element. If the flow engine is not stopped, the various entries of the *StateTask* data element (configuring the state machine) are retrieved for this workflow. Then, the engine loops through the various state tasks or transitions. For every state transition specified for this workflow element, all instances of the target data element are retrieved whose status corresponds to the begin state specified in the state transition. In a second (embedded) iteration, the flow engine loops through every instance of the data element and invokes the task element (`<Task>`) as specified in the state transition entry. In accordance with the internal structure of the task element, this corresponds to an invocation of the `perform` method of the `<Task>Bean` who will delegate the actual implementation to a delegation class. For every execution of a task on an instance of the data element, an entry is created in a `<Data>TaskStatus` data element for the purpose of logging and history tracking. As can be observed in the sequence diagram, the entry is created before the task is executed, and updated after execution (e.g., adding the timestamp at completion, and specifying whether the result was a success or failure).

B. Additional Requirements

Certain requirements for highly demanding back-end processes, related to data integrity and high performance, were not provided out-of-the-box by the above described initial flow element. However, such requirements are necessary when, for example, processing millions of income or VAT tax declarations. Consider for instance the need for locking or claiming of instances of the target data element, the need for the transactional encapsulation of the task execution and the setting of the result state, and the need for the parallel and simultaneous execution of a task on multiple instances of the target data element. An improved version of the flow element, incorporating these functionalities, should however still allow the regeneration of all existing (i.e., previously developed) applications, and therefore the regeneration of all existing instances of flow elements across all these applications. Therefore, backward compatibility is considered crucial: the default behavior of the improved flow element needs to correspond to the behavior of the original flow elements that are being regenerated or rejuvenated.

IV. Improved Flow Element Pattern

The flow element pattern was extended and improved in order to accommodate the additional functional requirements for high throughput processing as discussed Section III-B. In this section, we will discuss how the structure of the flow element pattern evolved, gradually introducing the additional functionality that satisfies the various functional requirements related to this high throughput processing.

A. Serial Instance Processing

After retrieving and applying the control data for the workflow engine, and following the retrieval of the various state transition entries for the workflow engine, the first additional piece of functionality is represented in the sequence diagram of Figure 3. In some cases, it is desired to process all tasks (or a number of them) in a serial or consecutive way on a single instance of the target data element. Consider for instance the consecutive processing of various tasks on a single invoice (e.g., computing the invoice, entering the in an accounting system, rendering the invoice document, and mail the invoice), instead of performing every first, second, etc. individual task on all invoices before processing the next task. For this purpose, a *StateTaskChainBuilder* is used to group the various state task transitions in consecutive chains. A parameter specifying the chain building strategy (which might for instance specify a maximal length for the chains) allows us to perfectly emulate the old behavior by indicating a maximal chain length equal to 1. The iteration to retrieve all instances of the target data element based on a specified begin state now loops over the various begin states of the state task chains, instead of using all begin states of all individual state tasks.

B. Claiming for Data Integrity

The second additional piece of functionality is the need for a data claiming mechanism. Before the actual processing of tasks on instances of the target data element, these instances are claimed using a claiming table in the database. Though it was already impossible for two different engines running in parallel to process the same instance of a target element simultaneously (through the use of intermediate states), processing time could be lost by using several parallel engines. Indeed, various parallel engines retrieving the same data instances frequently experienced during processing that data instances were already being processed by another engine in the initial version of the flow pattern. The mechanism of claiming data instances before processing them, avoids this.

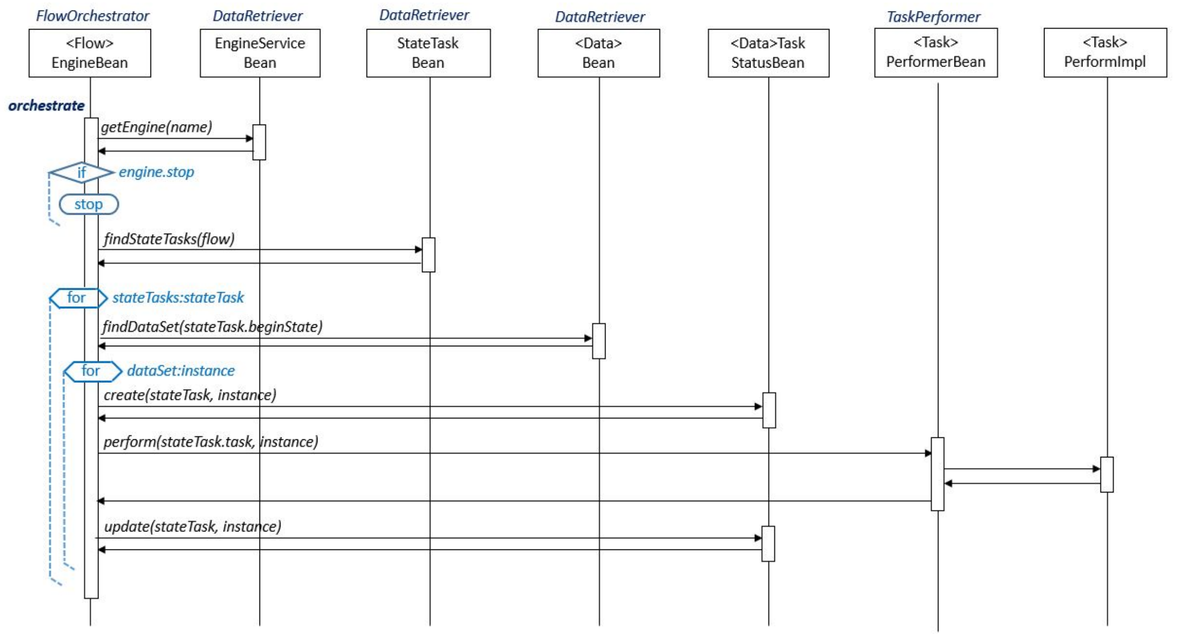

Figure 2. A sequence diagram representing the control flow of the initial flow element pattern.

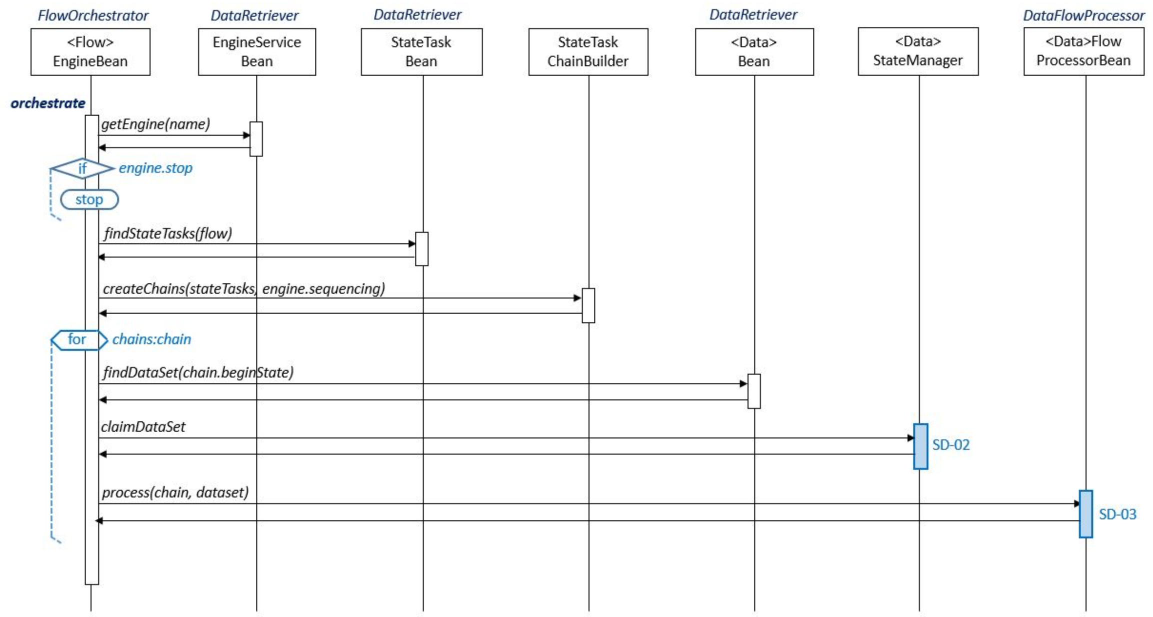

Figure 3. A sequence diagram containing the creation of state task chains and the claiming of instances in the improved flow element pattern.

C. Parallelized Task Processing

Now that a specific set of data instances has been claimed, they can be processed. Instead of looping straight away through all instances, the whole data set is passed to a `<Flow>Processor`, which is able to perform a single task —or a chain of consecutive tasks— in parallel threads simultaneously on multiple instances of the target data element. Note that backward compatibility can be achieved by using a parameter to specify the maximum amount of concurrent threads. This enables the emulation of the old behavior, by simply setting the value of the parameter to 1. Or alternatively, by using another `<Flow>Processor` implementation, which might even not allow parallel processing.

The control flow for this parallel processing mechanism is represented schematically in Figure 4. The parallel processor retrieves first the above mentioned parameter for the task(s) that are being processed, and starts the iteration through the various data element instances. For every instance, it is checked whether this instance has indeed been claimed by this engine, and a local hash map is retrieved to verify the maximum amount of instances that are currently being processed. If this maximum amount is reached, the flow engine will wait until a

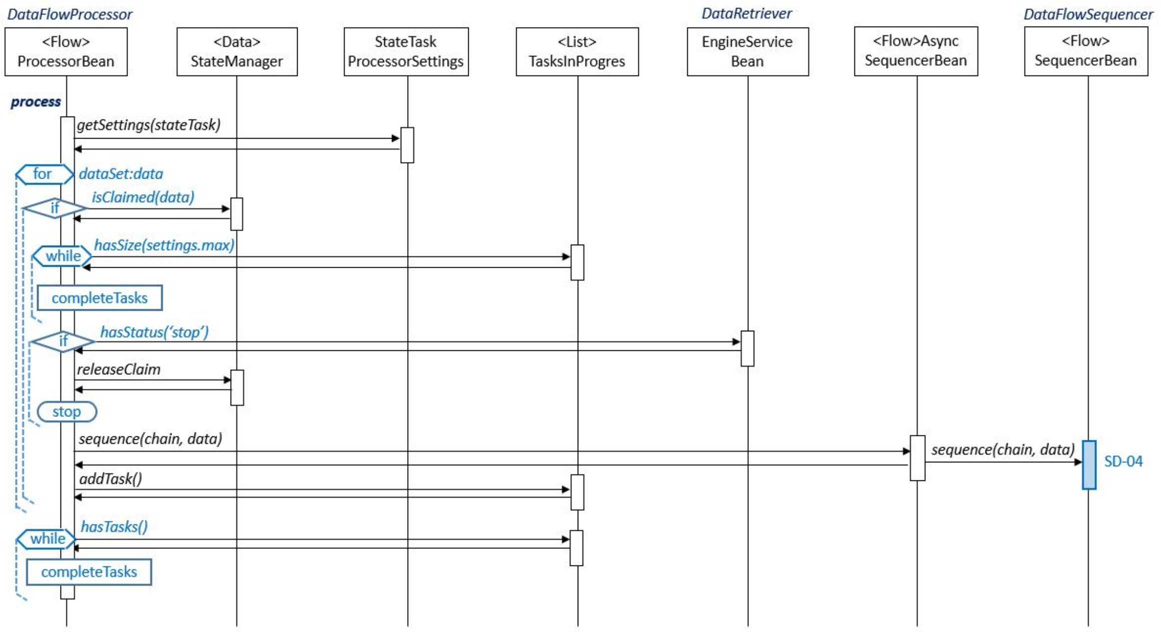

Figure 4. A sequence diagram containing the simultaneous and parallel processing of instances in the improved flow element pattern.

processing slot becomes available. If a slot becomes available, and the flow engine has not been ordered to stop in the control table in the meantime, a `<Flow>AsyncSequencer` is invoked to create a new processing thread. Within this newly created thread, which is being added to the hash map of running processing task threads, a `<Flow>Sequencer` is invoked to perform the (chain of) task(s) on the data instance. When processing threads have been launched for all instances of the data set, the `<Flow>Processor` simply waits until all processing threads have been completed.

D. Transactional State Transitions

Within the `sequence` method of the `<Flow>Sequencer` class, as represented in Figure 5, the data element instance is processed by the different consecutive tasks of the state task chain. For every task of the state task chain, three transactional steps ensure the transactional integrity of the statuses. First, the status of the data instance is set to an interim state, and an entry in the `<Data>TaskStatus` data element is created. Second, the target data instance is processed through an invocation of the actual task element, the status of the data instance is set to an end state, and the entry in the `<Data>TaskStatus` data element is updated (all within a single transaction). Finally, in case of failure, the setting of the end state and the updating of the entry in the history table, is combined with the creation of an additional entry in a failure history data element.

Also here it is crucial that the previously existing behavior of the flow element can be emulated by default. This is done by adding a transaction attribute to the specification of the data element. If not specified, a default value of *no_transaction* is selected, which omits the whole transactional behavior.

V. Empirical Test Results

The initial flow element has been generated by the NS expanders hundreds of times and was included in tens of software applications. Nearly all these applications —and therefore nearly all the corresponding flow elements— have been regenerated or rejuvenated multiple times during the last few years, with limited changes to the flow element pattern.

The new enhanced architecture of the flow element as described in this paper, and its implementation in the expanders, has been developed using a specific reference (test) application. Here, it was verified that the proposed solutions for transactional integrity performed as desired during system crashes. Moreover, it was validated via the reference application that a default set of parameters actually resulted in a behavior identical to the previous implementation of the flow element.

After this testing and validation 8 additional applications (containing a total of 31 flow elements) were regenerated using the new version of the NS expanders (containing the improved flow element pattern). Therefore, from that point in time, all 8 applications could make use of the additional requirements incorporated in the flow element, if preferred. These applications range from administrative applications (e.g., supporting the master thesis assessment of students or processing VAT tax declarations), to more industrial applications (e.g., supporting data hubs for energy providers or monitoring photovoltaic solar panels). The existing default behavior was tested in all 8 applications, whereas the new enhanced functionality was applied within two applications. Already one of these applications has been put into production (i.e., is being used) while running the improved flow element pattern.

VI. Conclusion

By discussing a specific NS element pattern and its evolution (resulting in the evolutionary enhancement of sets of software applications), this paper is believed to make several contributions. With regard to theory, we showed how the NST approach can be used for the simultaneous introduction of enhanced capabilities in (large) sets of information systems.

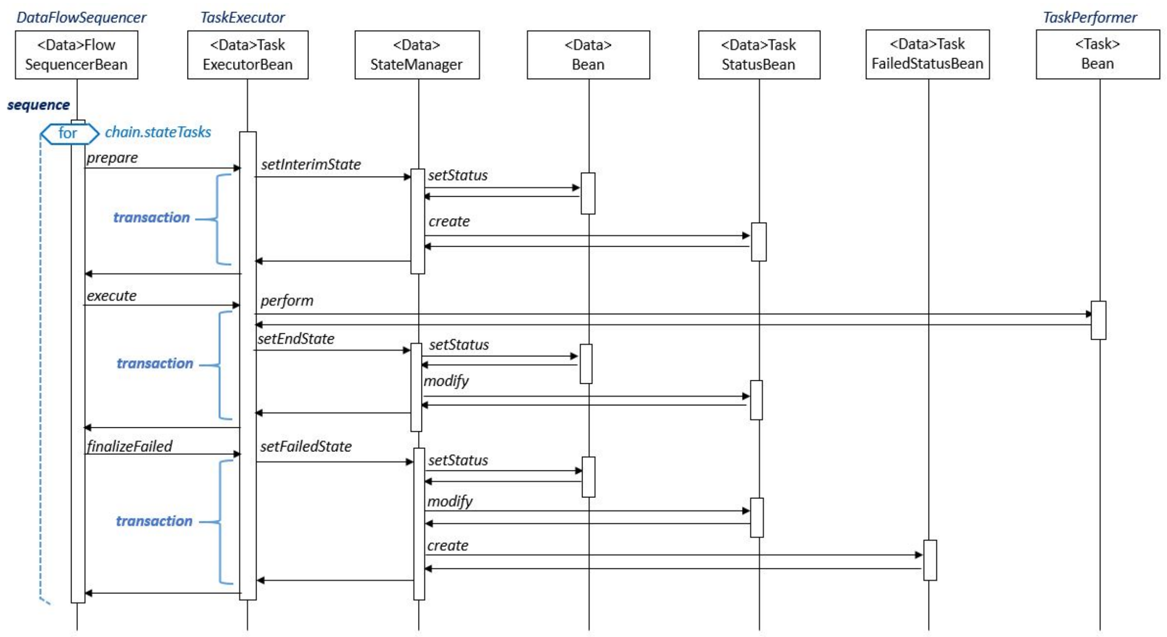

Figure 5. A sequence diagram containing the sequencing and transactional task processing in the improved flow element pattern.

This proves the feasibility of regenerating or rejuvenating (large) sets of information systems, and therefore enhancing their capabilities and/or modernizing the technologies used, while at the same time preserving existing functionality. For practitioners, this paper contributes to the design and documentation of actual workflow engines, processing state machines in a demanding high throughput environment.

Our paper has some limitations as well. First, the described set of information systems that has been regenerated and tested, is still somewhat limited, and only one regenerated application has been put into production using the enhanced pattern. Second, the proposed pattern for high throughput processing by state machine workflow engines has been designed and tested by several (but a limited amount of) experienced software developers, but has not yet been validated by a large amount of experts. A more advanced validation is however the ultimate goal of the NST approach: to validate and improve various software patterns through the collaborative efforts of many experts, and regenerate thousands of applications using these improved patterns. Third, our current discussion was mainly performed by means of a high-level overview (e.g. by using and describing sequence diagrams). A formal representation of the precise impact of using the (updated) workflow element, and the degree to which it is able to avoid the occurrence of combinatorial effects, is not provided in this work. Such further validations, improvements and formal representations are therefore considered as part of future research.

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.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional, 1994.

[12] T. Stahl, M. Völter, J. Bettin, A. Haase, and S. Helsen, Model-Driven Software Development. Wiley, 2006.