DESIGNING ENTERPRISE ARCHITECTURES BASED ON SYSTEMS THEORETIC STABILITY

Philip Huysmans, David Bellens, Dieter Van Nuffel and Kris Ven
Department of Management Information Systems, University of Antwerp, Prinsstraat 13, B-2000 Antwerp, Belgium

Keywords: Normalized systems, Design science research, Enterprise architecture, Method engineering.

Abstract: Contemporary organizations are operating in increasingly volatile environments. Hence, organizations must be agile in order to be able to quickly adapt to changes in its environment. Given the increasing complexity of organizations, it has been argued that organizations should be purposefully designed. Enterprise architecture frameworks provide guidance for the design of organizational structures. Unfortunately, current enterprise architecture frameworks have a descriptive, rather than a prescriptive nature and do not seem to have a strong theoretical foundation. In this paper, we explore the feasibility of extending the prescriptive design principles of the Normalized Systems theory to the field of enterprise architecture. Our results show that such approach is feasible and illustrate how the systems theoretic concept of stability can be used on the organizational level.

1 INTRODUCTION

Contemporary organizations are operating in increasingly volatile environments. Hence, organizations must be agile in order to be able to quickly adapt to changes in their environment. This may be a complex process, since a change to one organizational unit may affect other units. Given the increasing complexity of organizations, it has therefore been argued that organizations should be purposefully *designed* in order to exhibit true agility (Hoogervorst, 2009). Enterprise architecture frameworks support the design of the organizational structure, its business processes and information systems through a coherent set of principles, methods and models (Bernus et al., 2003). Unfortunately, current enterprise architecture frameworks have a descriptive, rather than a prescriptive nature. In order to purposefully design organizations, prescriptive principles are needed.

In software engineering literature, the Normalized Systems approach has recently been proposed to provide such deterministic design principles for the modular structure of software. The Normalized Systems approach is based on the systems theoretic concept of stability to ensure the evolvability of information systems. It argues that the main obstacle to evolvability is the existence of *combinatorial effects*. Combinatorial effects occur when the effort to apply a specific change increases as the system grows. This is a result of Lehman's law, which states: "*As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.*" (Lehman, 1980). The Normalized Systems approach eliminates these combinatorial effects by defining clear and deterministic design principles. Adhering to these principles therefore results in software systems that exhibit stability.

In this paper, we extend the Normalized Systems approach to the domain of enterprise architecture. The issue of combinatorial effects has not previously been explored in enterprise architecture frameworks, but seems relevant. Also, applying systems theory to the construction of organizations would advance the emerging field of enterprise engineering (Liles et al., 1995). We therefore explore the feasibility of designing a method for the construction of enterprise architectures that exhibit systems theoretic stability by eliminating combinatorial effects. Such a method would provide a more deterministic way of designing agile organizations. This method is currently being developed by using the design science methodology.

2 ENTERPRISE ARCHITECTURE

When market threats, opportunities or changes arise, the organization as a whole has to adapt. In order to be able to comprehend and manage the complexity of modern organizations, enterprise architecture frameworks have been introduced. Despite the common

Huysmans P., Bellens D., Van Nuffel D. and Ven K. (2010).
DESIGNING ENTERPRISE ARCHITECTURES BASED ON SYSTEMS THEORETIC STABILITY.
In *Proceedings of the International Conference on e-Business*, pages 157-162
DOI: 10.5220/0002985901570162

goal of enterprise architectures, many different frameworks are available. Various authors (e.g., (Leist and Zellner, 2006)) have compared these frameworks and identified differences and similarities. According to Leist and Zellner, who evaluated enterprise architecture frameworks with regard to the requirements of method engineering, no framework exists which provides all necessary elements to constitute a complete method (Leist and Zellner, 2006). Should an enterprise architect require the use of all elements, several (complementary) frameworks can be used concurrently, or a particular framework can be extended with missing elements. However, by combining or extending existing frameworks, the issue of *integration* between the models in the framework becomes even more complex. While most frameworks reduce the inherent complexity of an organization by offering separate views, it is not always clear how these views relate to or affect each other. The integration between the conceptual models should facilitate the translation of a single change in the outside world to all the different aspects of the organization.

However, if a change in a certain model affects other models it is combined with, a *combinatorial effect* occurs. While originally used in the Normalized Systems approach to describe evolvability in software, combinatorial effects also seem to affect evolvability on the enterprise architecture level. Analogously with combinatorial effects on the software level, this implies that organizations would become less evolvable as they grow. While the issue of integration has been acknowledged by other authors (e.g., (Lankhorst, 2005)), it has, to our knowledge, not yet been studied based on system theoretic concepts such as stability. By applying the design principles from Normalized Systems to enterprise architecture, we attempt to introduce these concepts in this field. In this paper, we elaborate on the construction of the *core diagram*. The core diagram is a model which provides an overview of the organizational scope which will be designed (Ross et al., 2006). Moreover, the core diagram aids understandability and communication of an enterprise architecture framework.

3 THEORETICAL FOUNDATION

3.1 Enterprise Ontology

Enterprise Ontology views the organization as a social system (Dietz, 2006). Therefore, it is well suited to describe the interaction between an organization and its environment. Enterprise Ontology assumes that communication between human actors is a necessary and sufficient basis for a theory of organizations (Dietz, 2006). This is based on the language action perspective and Habermas theory of communicative action. The strong theoretical foundation ensures a consistent modelling methodology. Clear guidelines are provided to create abstract models. Since only the *ontological* acts are represented in the models, the same model will be created for organizations who perform the same function, but operate differently. For example, consider the BPR case at Ford (Hammer, 1990). The ontological model of the processes of the situation before and after reengineering are identical. Because of the focus on the essential business processes, Enterprise Ontology models can be very concise. Therefore, they provide a good overview of a broad enterprise scope, and are well suited as an enterprise architecture core model.

The transaction pattern describes the coordination necessary to produce a certain result. This result is represented by a *production fact*. There are always two actors involved in a transaction: the *initiator* actor who wants to achieve the fact, and the *executor* actor who performs the necessary actions to create the fact. Delivering a product, performing a service or subscribing to an insurance are examples of production facts which could be created by completing a transaction. The high-level structure of the transaction pattern consists of three phases. In the order phase, the actors negotiate the subject of the transaction. In the execute phase, the subject of the transaction is brought about. In the result phase, the result of the transaction is presented and accepted. In different versions of the transaction pattern, different ontological process steps are identified in the three phases. These steps are called *coordination acts*. The successful completion of an act results in a *coordination fact*.

The basic transaction pattern consists of the five standard acts which occur in a successful scenario (i.e., request, promise, execute, state and accept) (Dietz, 2006, p. 90). Consider a transaction in the case of a simple product delivery process. In the order phase, the customer *requests* the product. Once this request is adequately specified, the request coordination fact is created. The supplier then *promises* to deliver the product according to the agreed terms. This creates the promise coordination fact. In the execute phase, the executor actually performs the the execute act, resulting in the production fact. In our example, this is the actual delivery (i.e., “Product X has been delivered”). In the result phase, the supplier *states* that the delivery has been completed. If the customer is satisfied with the delivery, he will accept the delivery in the *accept* process step. Once the accept coordina-

tion fact is created, the transaction is considered to be completed.

The *standard* transaction pattern is the basic transaction pattern, augmented with the scenario in which the actors dissent (Dietz, 2006, p. 93). In the order-phase, the executor actor can decline the incoming request of the initiator actor. The initiator then has to decide whether he resubmits his request, or quits the transaction. In our example, the supplier could decline the delivery of a product which does not belong to his catalogue. The customer would need to select another product, or quit the transaction and search another supplier. The execute-phase is identical to the execute-phase in the basic transaction pattern. In the result-phase, the initiator actor can reject the stated production fact instead of accepting it. The executor then has to decide whether he wants to repeat the execution act and make the statement again, or stop the transaction.

3.2 Normalized Systems

The basic assumption of the Normalized Systems approach is that information systems should be able to evolve over time, and should be designed to accommodate change. To genuinely design information systems accommodating change, they should exhibit *stability* towards requirements changes. In systems theory, stability refers to the fact that bounded input to a function results in bounded output values, even as $t \rightarrow \infty$. When applied to information systems, this implies that no change propagation effects should be present within the system; meaning that a specific change to an information system should require the same effort, irrespective of the information system's size or the point in time when being applied. *Combinatorial effects* occur when changes require increasing effort as the system grows. They need to be avoided in stable systems. Normalized Systems are therefore defined as information systems exhibiting stability with respect to a defined set of changes (Mannaert and Verelst, 2009), and are as such defying Lehman's law of increasing complexity (Lehman, 1980) and avoiding the occurrence of combinatorial effects.

The Normalized Systems approach proposes a set of four *design principles* that act as design rules to identify and circumvent most combinatorial effects (Mannaert and Verelst, 2009). The first principle, *separation of concerns*, implies that every change driver or concern should be separated from other concerns. This theorem allows for the isolation of the impact of each change driver. The second principle, *data version transparency*, implies that data should be communicated in version transparent ways between components. This requires that this data can be changed (e.g., additional data can be sent between components), without having an impact on the components and their interfaces. The third principle, *action version transparency*, implies that a component can be upgraded without impacting the calling components. This principle can be accomplished by appropriate and systematic use of, for example, polymorphism or a facade pattern. The fourth principle, *separation of states*, implies that actions or steps in a workflow should be separated from each other in time by keeping state after every action or step. This suggests an asynchronous and stateful way of calling other components.

The design principles show that software constructs, such as functions and classes, by themselves offer no mechanisms to accommodate anticipated changes in a stable manner. The Normalized Systems approach therefore proposes to encapsulate software constructs in a set of five higher-level software elements. These elements are modular structures that adhere to these design principles, in order to provide the required stability with respect to the anticipated changes (Mannaert and Verelst, 2009). From the second and third principle it can straightforwardly be deduced that the basic software constructs, i.e., data and actions, have to be encapsulated in their designated construct. As such, a *data element* represents an encapsulated data construct with its get- and set-methods to provide access to their information in a data version transparent way. So-called cross-cutting concerns, for instance access control and persistency, should be added to the element in separate constructs. The second element, *action element*, contains a core action representing one and only one functional task. Four different implementations of an action element can be distinguished: *standard* actions, *manual* actions, *bridge* actions and *external* actions. In a standard action, the actual task is programmed in the action element and performed by the same information system. In a manual action, a human act is required to fulfil the task. The user then has to set the state of the life cycle data element through a user interface, after the completion of the task. A process step can also require more complex behaviour. A single task in a workflow can be required to take care of other aspects, which are not the concern of that particular flow. Therefore, a separate workflow will be created to handle these concerns. Bridge actions create these other data elements going through their designated flow. When an existing, external application is already in use to perform the required task, the action element would be implemented as an external action.

These actions call other information systems and set their end state depending on the external systems' reported answer. Based upon the first and fourth principle, workflow has to be separated from other action elements. These action elements must be isolated by intermediate states, and information systems have to react to states. To enable these prerequisites, three additional elements are identified. A third element is thus a *workflow element* containing the sequence in which a number of action elements should be executed in order to fulfill a flow. A consequence of the stateful workflow elements is that state is required for every instance of use of an action element, and that the state therefore needs to be linked to or be part of the instance of the data element serving as argument. A *trigger element* is a fourth one controlling the states (both regular and error states) and checking whether an action element has to be triggered. Finally, the *connector element* ensures that external systems can interact with data elements without allowing an action element to be called in a stateless way.

4 ARTEFACT CONSTRUCTION

In this section, we outline the construction of a core diagram which is based on Enterprise Ontology models and expressed in Normalized Systems constructs. Since Enterprise Ontology models are implementation-independent, we can base our method on these models to implement the needed organizational aspects in the transactions. The resulting artefact is called a *Normalized Systems Business Transaction (NSBT)*. In order to illustrate the different steps, we use a mail order example. In this example, different implementations of ontological process steps are available. For example, consider the request of the order. Instead of using a standard mail form, the company can offer the customer the possibility to place the order on a website. We introduce these variations to illustrate the evolvability of the NSBT with regard to changes in implementation technology. While this changes the implementation, no changes are made to the *essential* Enterprise Ontology models.

In the mail order example, an Enterprise Ontology transaction would result in a production fact "*an order has been delivered*". In Normalized Systems, this transaction pattern is represented by a flow element. A flow element is driven by precisely one data element, the life cycle data element. In order to define a Normalized Systems flow, we thus need a `T01` data element. The completion of the different acts in the transaction process is represented by the creation of ontological facts. In Normalized Systems, these facts are represented by the states which occur in the flow element, being the life cycle states of the corresponding data element. To reach these states, a state transition is required. A state transition is realized by an action element. The successful completion of that action element results in the defined life cycle state. In order to define the control flow of the process, we therefore need to specify the *trigger states*, *state transitions* and *transaction actions*. Regarding the request coordination fact, this implies that the T01 flow element, and thus also the corresponding T01 data element, should reach the state *Requested*. This means that upon initiation of a T01 transaction, a new T01 data element is instantiated, resulting in the life cycle state *Initial*. The genuine act of requesting is encapsulated in the action element `Request`. In the mail order example, the `Request` action element would contain the task which ensures that the order request is fully defined by the customer. In our automated version of the example, this functionality is offered by the website. However, if the traditional mail order request form needs to be supported as well, a second `Request` action element could be created. This action element would be implemented as a manual action. When a retail company employee receives the order request form, the *Requested* state will be set manually. The remainder of the transaction will be handled identically, regardless of the implementation method of the request. The *concerns* of creating the data element and handling the request are thus separated as they can clearly evolve independently from each other.

While all state transitions are defined as action elements, their different nature can mean that they need to be implemented differently. Consider the notification of the initiator actor in the promise process step. If this notification requires a human action, the `Promise` action element would be implemented as a manual action. For example, it could be that the order request needs to be approved by an employee of the retail company. However, the promise process step can also require more complex behaviour. When for example the requested product first needs to be reserved in the warehouse, the `Promise` action element would be implemented as a bridge action triggering a flow element on another data element, e.g., a `Reservation` element. When the retail company already has an existing application in use to perform these reservations, the `Promise` action element would be implemented as an external action.

However, the transaction process does not always follow the successful scenario. In the scenario in which the actors can dissent, additional coordination acts need to be added. When translating these acts to

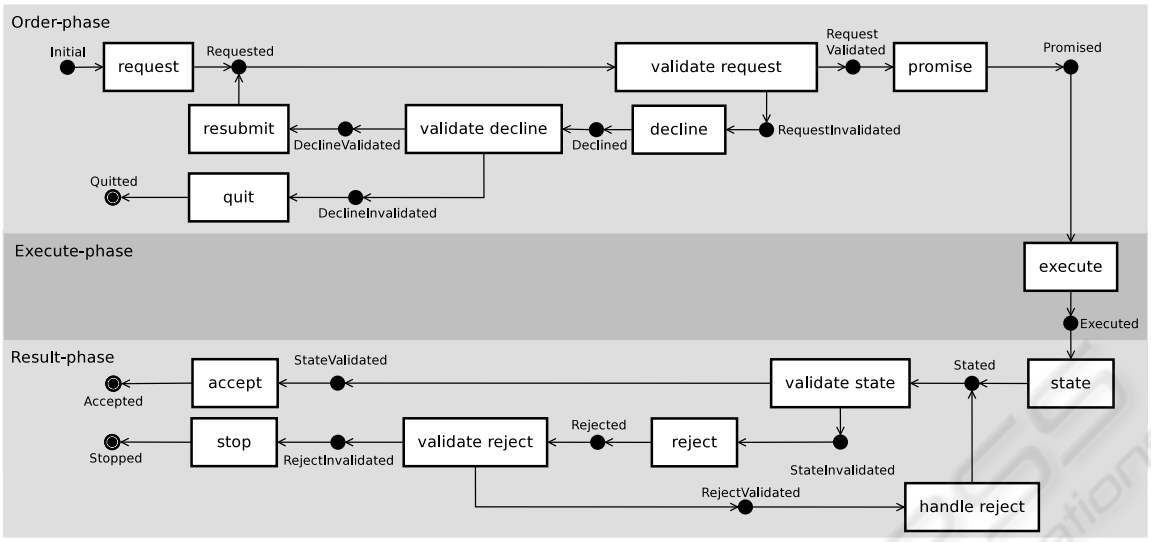

Figure 1: Graphical representation of the standard transaction pattern flow.

Normalized Systems primitives, some additional actions and states have to be included due to the Normalized Systems theorems. The resulting Normalized Systems flow element is graphically represented in Figure 1. Based on *separation of concerns*, the decision of the executor actor to promise or decline the request needs to be separated from the actual coordination act (i.e., the communication of the decision). The decision logic to promise or decline can change independently from the communication method, as shown by the various implementations of the `Promise` action element described above. Since the communication method can also change independently of the decision logic, these two actions should not be combined in one action element. Doing so would introduce a combinatorial effect. Therefore, we introduce an additional action element `ValidateRequest`. In the case where the executor decides to handle the request, the state *RequestValidated* is set. Otherwise, the state *RequestInvalidated* is set. The actual `Promise` action element then contains the actual communication of the decision. In our example, the `ValidateRequest` action element can contain the logic to check whether the retail company can deliver the order, e.g., whether the product is in stock. If the request is declined, the initiator actor needs to decide whether or not to resubmit the request. This decision logic is again separated from the other actions by encapsulating the decision logic in an action element `ValidateDecline`. If the initiator decides to resubmit, the state is set to *DeclineValidated*. The `Resubmit` action element then allows the initiator actor to possibly change the request and to resubmit it which will again result in the state *Requested*. If the product from the original order request is not available, the customer thus has the option to adapt his order and resubmit it. If the initiator decides to abort the transaction, the state is set to *DeclineInvalidated*, which triggers the `Quit` action element to reach the end state *Quitted*. Analogously, the initiator actor has to decide whether he accepts the stated production fact in the result phase of the transaction. We therefore introduce the `ValidateState` action element, which results in the *StateValidated* state in case of a successful acceptance, or in the *StateInvalidated* state in case of an unsuccessful one. The *StateValidated* state triggers the `Accept` action element, which contains the actual accept coordination act. In case the initiator does not accept the state coordination fact, the workflow is brought to the *Rejected* state through the `Reject` action element. The separation of concerns theorem forces us again to separate the action element containing the decision logic (i.e., the `ValidateState` action element) from the action element containing the communication method (i.e., the `Accept` and `Reject` action elements). In our example, it is possible that the customer is not satisfied with the delivered products. This decision can be implemented as a manual action element. The user would manually check the delivery, and indicate whether he accepts it. However, in the context of a B2B transaction, it could be that an automated quality control system is in place. In that case, an external action element would be used. These different action elements could bring the state of the transaction workflow in the *StateInvalidated* state. The `Reject` action element, which communicates the decision, can again be implemented using the different types of action elements. The decision whether to handle the reject is taken in the `ValidateReject` action element. The reject handling itself is implemented as a dedicated

`HandleReject` action element. If the retail company agrees with the complaint, it can modify the delivery and state the delivery again. If the executor does not handle the reject, the transaction reaches the end state *Stopped* through the `Stop` action element.

5 DISCUSSION AND CONCLUSIONS

In this paper, it has been shown that an NSBT, which consists only of Normalized Systems elements, can be constructed as a core diagram for an enterprise architecture framework. By using the combination of implementation-independent Enterprise Ontology models and evolvable Normalized Systems elements, we have demonstrated the flexibility in an e-business example, by adding a multi-channel request process. Should the request of the delivery transaction not have been separated from the execution (i.e., the actual delivery), such an addition would have resulted in a combinatorial effect. Consider the following situation, in which this separation has been neglected: separate delivery systems are used for physical requests (i.e., which are requested through mail) and electronical requests (i.e., orders from the website). A simple change in the delivery execution, such as recording the delivery of a certain product, would require two distinct implementations. The impact of the change would thus depend on the size of the system: when more retail channels would be added, the same change would cause an even larger impact. Such combinatorial effects should be avoided. In subsequent iterations, we will integrate other aspects present in enterprise architecture frameworks. This will be done analogously to the integration of cross-cutting concerns on the software level into Normalized Systems elements.

This paper has two important contributions. A first contribution is that we introduced the concept of combinatorial effects on the level of enterprise architectures. We further illustrated how the systems theoretic concept of stability can be applied to the design of enterprise architectures. This requires the elimination of combinatorial effects, which will lead to more evolvable organizations. As a result, we offer a view on enterprise agility that has a strong theoretical foundation. A second contribution is that we demonstrated the feasibility of constructing an enterprise architecture core diagram based on existing scientific approaches. By expressing the core diagram in Normalized Systems elements, we extended the Normalized Systems approach to the organizational level. Using Enterprise Ontology models as the basis for the core diagram further demonstrates the feasibility of constructing an enterprise architecture framework based on scientific theories. This illustrates how theories from relevant fields can be applied in a new setting by using a design science approach.

REFERENCES

Bernus, P., Nemes, L., and Smidth, G. (2003). *Handbook on Enterprise Architecture*. International Handbooks on Information Systems. Springer-Verlag, Berlin.

Dietz, J. L. (2006). *Enterprise Ontology: Theory and Methodology*. Springer, Berlin.

Hammer, M. (1990). Reengineering work: Don't automate, obliterate. *Harvard Business Review*, 68(4):104ff.

Hoogervorst, J. A. P. (2009). *Enterprise Governance and Enterprise Engineering (The Enterprise Engineering Series)*. Springer, 1st edition.

Lankhorst, M. M. (2005). Enterprise architecture modelling–the issue of integration. *Advanced Engineering Informatics*, 18(4):205 – 216. Enterprise Modelling and System Support.

Lehman, M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68:1060–1076.

Leist, S. and Zellner, G. (2006). Evaluation of current architecture frameworks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1546–1553, New York, NY, USA. ACM.

Liles, D. H., Johnson, M., Meade, L., and Underdown, D. (1995). Enterprise engineering: A discipline? In for Enterprise Engineering, S., editor, *Proceedings of the Society for Enterprise Engineering Conference*, Kettering, Ohio.

Mannaert, H. and Verelst, J. (2009). *Normalized Systems: Re-creating Information Technology Based on Laws for Software Evolvability*. Koppa, Hasselt.

Ross, J. W., Weill, P., and Robertson., D. C. (2006). *Enterprise Architecture as Strategy – Creating a Foundation for Business Execution.* Harvard Business School Press, Boston, MA.