Aligning the Constructs of Enterprise Ontology and Normalized Systems

Philip Huysmans, David Bellens, Dieter Van Nuffel, and Kris Ven

Department of Management Information Systems,
University of Antwerp, Antwerp, Belgium
{philip.huysmans,david.bellens,dieter.vannuffel,kris.ven}@ua.ac.be

Abstract. Literature suggests that, due to their complexity, organizations need to be designed in order to be effective and evolvable. Recently, two promising approaches have been introduced that are relevant in this regard. Enterprise Ontology creates essential models that are implementation-independent. Normalized Systems is concerned with the development of information systems with proven evolvability. In this paper, we combine both approaches. To this end, we express the transaction pattern—a central construct of Enterprise Ontology—using the constructs of Normalized Systems. By aligning these constructs, we attempt to introduce traceability between the Enterprise Ontology level and the Normalized Systems level. The resulting artefact exhibits the benefits of both Enterprise Ontology and Normalized Systems. We illustrate the application of the artefact in the context of enterprise architectures.

Keywords: Enterprise Ontology, Normalized Systems, Enterprise Architecture.

1 Introduction

Contemporary organizations have to be agile in order to be able to adapt to changing market environments. A change of the organization as a whole affects many different organizational elements. Given the complexity of organizations, it can be argued that organizations should be *designed* in order to exhibit true agility [7]. Enterprise architecture is proposed as a way to control this complexity. Despite the multitude of frameworks available, no common scientific or theoretical foundation seems to be agreed upon. Therefore, it is difficult to compare and evaluate the recommendations made by these frameworks [10]. In this paper, we explore an approach which focuses on the organizational ability to change. We base our approach on the systems theoretic concept of evolvability by applying the theorems of Normalized Systems. By adhering to the four theorems of Normalized Systems during software design and development, software architectures of proven evolvability are obtained [11]. Based on these theorems, Normalized Systems proposes five software elements to design the modular structure of software. This modular structure ensures that the software is free from so-called combinatorial effects. Enterprise Ontology provides abstract, implementation-independent organizational

A. Albani and J.L.G. Dietz (Eds.): CIAO! 2010, LNBIP 49, pp. 1–15, 2010.

constructs which can describe a broad organizational scope with few construct instantiations (i.e., transactions) [2]. In order to work towards evolvable organizations, we explore an implementation of Enterprise Ontology transactions which are evolvable. Normalized Systems suggests that we therefore need an implementation which is free of combinatorial effects. Different alternatives seem to be available to reach this goal. A first alternative would be to define an implementation which closely mimics the structure of the transaction pattern, and ensure that this implementation is free of combinatorial effects. While a complete specification of the implementation is not possible on this abstraction level, we could base further specification upon this implementation. A second alternative would be to implement a completely specified Enterprise Ontology model, in which we could then eliminate the combinatorial effects. In this paper, we explore the first alternative. More specific, we explore the expression of the transaction pattern—a core Enterprise Ontology construct—in Normalized Systems elements. While similar, more practice-driven approaches may exist, we limit ourselves to the combination of Enterprise Ontology and Normalized Systems, because of their scientific foundation.

The rest of the paper is structured as follows. In Section 2, we introduce Normalized Systems and Enterprise Ontology. We then describe the alignment of the constructs in Section 3. Next, We position our approach within enterprise architectures in Section 4. Finally, we offer our conclusions in Section 5.

2 Scientific Foundations

In this section, we provide a brief introduction to both Enterprise Ontology and Normalized Systems. We primarily focus on the constructs of both approaches that were used in our research.

2.1 Enterprise Ontology

Theoretical Foundation. In order to grasp the complexity of organizations, models can be constructed. These models abstract away from the information that is available in the real world. Depending on the way of abstraction, very diverse models can be made. Enterprise Ontology views the organization as a social system [2]. 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 [2]. 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 [6]. 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. Many case studies are reported where large organizations are described with few modelling artefacts (e.g. [13]).

In this paper, we will focus on the transaction pattern as the basic construct of Enterprise Ontology. We currently focus on the transaction as the main Enterprise Ontology primitive for two main reasons. First, the transaction pattern is a core element of the Enterprise Ontology theory. The transaction pattern is specified by the transaction axiom, the second axiom from the Ψ-theory on which Enterprise Ontology is based. Second, it is the basis on which other models elaborate. It therefore seems logical to base our initial effort on achieving a correct mapping of the transaction pattern.

The transaction pattern evolved thanks to contributions from many researchers [1,5,15,17]. 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.

Enterprise Ontology Artefact. 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*. Enterprise Ontology distinguishes between the basic, standard and complete transaction pattern.

The graphical representation of the transaction pattern is shown in Figure 1. The combination of a coordination act and fact is represented by a circle in a square. The combination of a production act and fact is represented by a diamond in a square. Small circles represent process entry points, and small circles with a cross represent a choice between alternate flows. Light-grey boxes indicate which acts fall under the authority of a certain actor. In Figure 1, A01 is the initiator actor, and A02 is the executor actor.

Basic Transaction Pattern: The basic transaction pattern consists of the five standard acts which occur in a successful scenario (i.e., request, promise, execute, state and accept) [2, p. 90]. These five acts are shown in the centre of Figure 1. In the order-phase, the initiator actor first *requests* the creation of a fact. The executor actor then *promises* to fulfil this request. In the execute-phase, the executor actually performs the necessary actions to create the fact in the *execute* act. In the result phase, the executor first *states* the successful completion of the fact. Finally, the initiator *accepts* this statement. Consider this transaction in the case of a simple product delivery process. In a first process step, the customer requests the product. Once this request is adequately specified, the request coordination fact is created. Second, the supplier promises to deliver the product according to the agreed terms. This creates the promise coordination fact. The third process

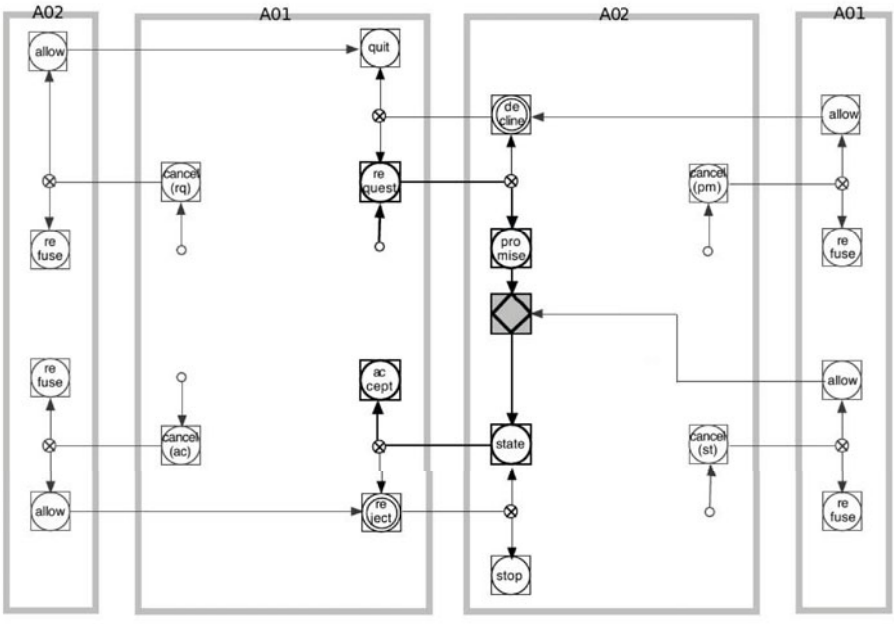

Fig. 1. Graphical representation of the Enterprise Ontology Transaction Pattern

step is the actual delivery. This results in the production fact "Product X has been delivered". In the fourth process step, the supplier states that the delivery has been completed. If the customer is satisfied with the delivery, he will accept the delivery in the fifth process step. Once the accept coordination fact is created, the transaction is considered to be completed.

Standard Transaction Pattern: The *standard* transaction pattern is the basic transaction pattern, augmented with the scenario in which the actors dissent [2, p. 93]. The coordination facts which indicate a dissent are represented by a double circle in Figure 1 (i.e., decline and reject facts). 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.

Complete Transaction Pattern: In the complete transaction pattern, cancelation patterns are added to the standard transaction pattern. In Figure 1, the cancelation patterns are started in the four additional process entry points. According

to [2], every coordination fact can be cancelled at any time by the responsible actor. This cancelation can then be allowed or refused by the other actor. For example, when the customer changes his mind after requesting the delivery of a product, he can cancel his request. The executor then has to decide whether he allows this cancelation, in which case the transaction ends, or refuses the cancelation, and proceeds with the transaction.

2.2 Normalized Systems

Theoretical Foundation. The basic assumption of Normalized Systems is that information systems should be able to evolve over time, and should be designed to accommodate change. As this evolution due to changing business requirements is mostly situated during the mature life cycle stage of an information system, it takes the form of software maintenance. Software maintenance is considered to be the most expensive phase of the information system's life cycle, and often leads to an increase of architectural complexity and a decrease of software quality [4]. This phenomenon is also known as Lehman's law of increasing complexity, expressing the degradation of information system's structure over time [9]. Because changes applied to information systems are suffering from Lehman's law, the impact of a single change will increase over time as well [8]. Therefore to genuinely design information systems accommodating change, they should exhibit *stability* towards these 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. This means that a specific change to an information system should require the same effort, irrespective of the information system's size or point in time when being applied. *Combinatorial effects* occur when changes require increasing effort as the system grows; and should thus be avoided. Normalized systems are defined as information systems exhibiting stability with respect to a defined set of changes [11], and are as such defying Lehman's law of increasing complexity [8,9] and avoiding the occurrence of combinatorial effects. In this sense, evolvability is operationalized as a number of anticipated changes that occur to software systems during their life cycle [12].

The normalized systems approach deduces a set of four *design theorems* that act as design rules to identify and circumvent most combinatorial effects [11,12]. It needs to be emphasized that each of these theorems is not completely new, and even relates to the heuristic knowledge of developers. However, formulating this knowledge as theorems that identify these combinatorial effects aids to build systems containing minimal combinatorial effects. The first theorem, *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. Parnas described this principle already in 1972 [14] as what was later called *design for change*. Applying the theorem prescribes that each module can contain only one submodular task (which is defined as a change driver), but also that workflow should be separated from functional submodular tasks.

The second theorem, *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. This theorem can, for example, be accomplished by appropriate and systematic use of web services instead of using binary transfer of parameters. This also implies that most external APIs cannot be used directly, since they use an enumeration of primitive data types in their interface.

The third theorem, *action version transparency*, implies that a component can be upgraded without impacting the calling components. This theorem can be accomplished by appropriate and systematic use of, for example, polymorphism or a facade pattern.

The fourth theorem, *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. Synchronous calls resulting in pipelines of objects calling other objects which are typical for object-oriented development result in combinatorial effects.

Normalized Systems Artefacts. The design theorems show that software constructs, such as functions and classes, by themselves offer no mechanisms to accommodate anticipated changes in a stable manner. Normalized Systems therefore proposes to encapsulate software constructs in a set of five higher-level software elements (i.e., data, action, flow, connector and trigger elements). These elements are modular structures that adhere to these design theorems, in order to provide the required stability with respect to the anticipated changes [11]. To map the Enterprise Ontology transaction pattern, three of these five higher-level software elements are needed. We will now elaborate on these three elements.

From the second and third theorem it can straightforwardly be deduced that the basic software constructs, i.e., data and actions, have to be encapsulated in their designated construct.

Data Element: 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.

Action Element: The second element, *action element*, contains a core action representing one and only one functional task. Arguments and parameters need to be encapsulated as separate data elements, and cross-cutting concerns like logging and remote access should be again added as separate constructs. [16] distinguish between four different implementations of an action element: *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. Consider the ordering of parts for an assembly. The assembly workflow needs to know when the parts are ready to be assembled, but it is not concerned with how the parts are prepared. Therefore, a separate workflow will be created to handle the concerns of the individual parts. Bridge actions create these other data elements going through their designated flow. Fourth, when an existing, external application is already in use to perform the actions on, for instance, the different parts of an assembly, 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.

Workflow Element: Based upon the first and fourth theorem, 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. A third element is thus a *workflow element* containing the sequence in which a number of action elements should be executed in order to fulfil 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 or be part of the instance of the data element serving as argument. We call this data element the life cycle data element of a flow. A graphical representation of a flow element is shown in Figure 2. This representation is consistent with the representation of Normalized Systems workflow elements, which are based on state machines [11, p. 143]. The black circles represent the different states of the flow, being the life cycles states of the corresponding data element. The state name is notated next to the state symbol. The squares represent the action elements.

3 Translating the Transaction Pattern

As discussed in Section 2.1, the transaction pattern of Enterprise Ontology is the starting point for our research. In this section, we present the mapping of the transaction pattern to the constructs of Normalized Systems, which are discussed in Section 2.2. We will start by translating the *basic* transaction pattern, and iteratively add more details in the *standard* transaction pattern and the *cancelation patterns*.

3.1 The Basic Transaction Pattern

We start by mapping the basic transaction pattern. The basic transaction pattern consists of the process steps request, promise, execute, state and accept. In Normalized Systems, this transaction pattern process is represented by a *flow* element. A flow element is driven by precisely one data element, the life cycle data element. Consider a transaction T01. 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 trough its default constructor, resulting in the life cycle state *Initial*. The genuine act of requesting is encapsulated in the action element `Request`. The *concerns* of creating the data element and handling the request are separated as they can clearly evolve independently from each other. The request could, for example, contain additional information that needs to be processed. Since we are currently only regarding the successful flow of the transaction, we do not yet need any branching. The state transition can be expected to always result in the end state *Requested*. The resulting Normalized Systems flow is shown in Figure 2, and schematically represented in Table 1.

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, e.g., a manager who has to decide, the `Promise` action element would be implemented as a manual action. However, the promise process step

Table 1. Specification of the basic transaction pattern flow element

Workflow name		Basic Transaction Pattern	
Data element		T01-basic	
Start state	**Action name**	**End state**	**Failed state**
Initial	Request	Requested	
Requested	Promise	Promised	
Promised	Execute	Executed	
Executed	State	Stated	
Stated	Accept	Accepted	

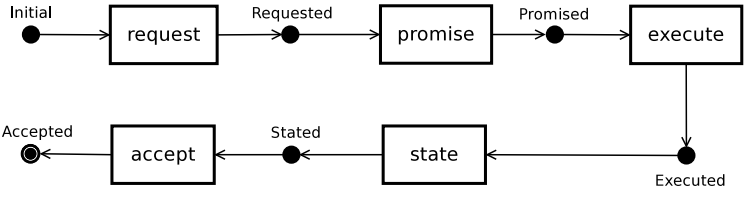

Fig. 2. Graphical representation of the basic transaction pattern flow

can also require more complex behaviour. When for example the 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 `Part` element. When an existing application is already in use to perform these reservations, the `Promise` action element would be implemented as an external action.

3.2 The Standard Transaction Pattern

The standard transaction pattern adds the scenario in which the actors can dissent. When translating these additions to 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 3. 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 communication method can change independently, as shown by the various implementations of the `Promise` action element in the basic transaction pattern. Since the decision logic to promise or decline can also change independently of the communication method, 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 remains identical to the action element described in the basic transaction pattern. If the request is however 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 initiator decides to abort the transaction, the state is set to *DeclineValidated*, 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. 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 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 executor does not handle the reject, the transaction reaches the end state *Stopped* through the `Stop` action element. All the described state transitions for the standard transaction pattern are summarized in Table 2.

Table 2. Specification of the standard transaction pattern flow element

Workflow name		Standard Transaction Pattern	
Data element		T01-standard	
Start state	**Action name**	**End state**	**Failed state**
Initial	Request	Requested	
Requested	ValidateRequest	RequestValidated	RequestInvalidated
RequestInvalidated	Decline	Declined	
RequestInvalidated	ValidateDecline	DeclineValidated	DeclineInvalidated
DeclineInvalidated	Quit	Quitted	
DeclineValidated	Resubmit	Requested	
RequestValidated	Promise	Promised	
Promised	Execute	Executed	
Executed	State	Stated	
Stated	ValidateState	StateValidated	StateInvalidated
StateInvalidated	Reject	Rejected	
Rejected	ValidateReject	RejectValidated	RejectInvalidated
RejectInvalidated	Stop	Stopped	
RejectValidated	HandleReject	Stated	
StateValidated	Accept	Accepted	

3.3 The Cancelation Patterns

The *complete* transaction pattern also includes the various cancelation patterns and is shown in Figure 1. A cancelation consists of two main issues: deciding whether or not to allow a cancel request and handling the cancelation itself. The first issue actually consists of initially receiving the cancel request, then deciding whether or not to allow the requested cancelation, and third potentially to notify the initiator of the rejected cancel request. As such, based on *separation of states* and *separation of concerns*, these three concerns will be separated. First, upon arrival of a cancel request, a dedicated `CancelRequest` data element will be created. This implies that for every life cycle data element that can be cancelled, a related `CancelRequest` data element instance will be created if such a request arrives. For example, for a life-cycle data element called `Order`, a corresponding `OrderCancelRequest` data element will be created. Second, an action element `AcceptCancelation` will implement the decision whether or not to accept. Third, in case of an rejected request, the initiator will probably have to be notified. This functionality is represented by a bridge action `Refuse` executing the notification in the way as discussed in [16]. In case of an allowed cancelation, the `CancelTransaction` standard action element will initiate the cancelation handling which will be explained next. The Normalized Systems specification for the workflow representing the cancel request issue is shown in Table 3 and Figure 4. In case of an allowed cancelation, `CancelTransaction` standard action element will initiate the handling itself explained hereafter.

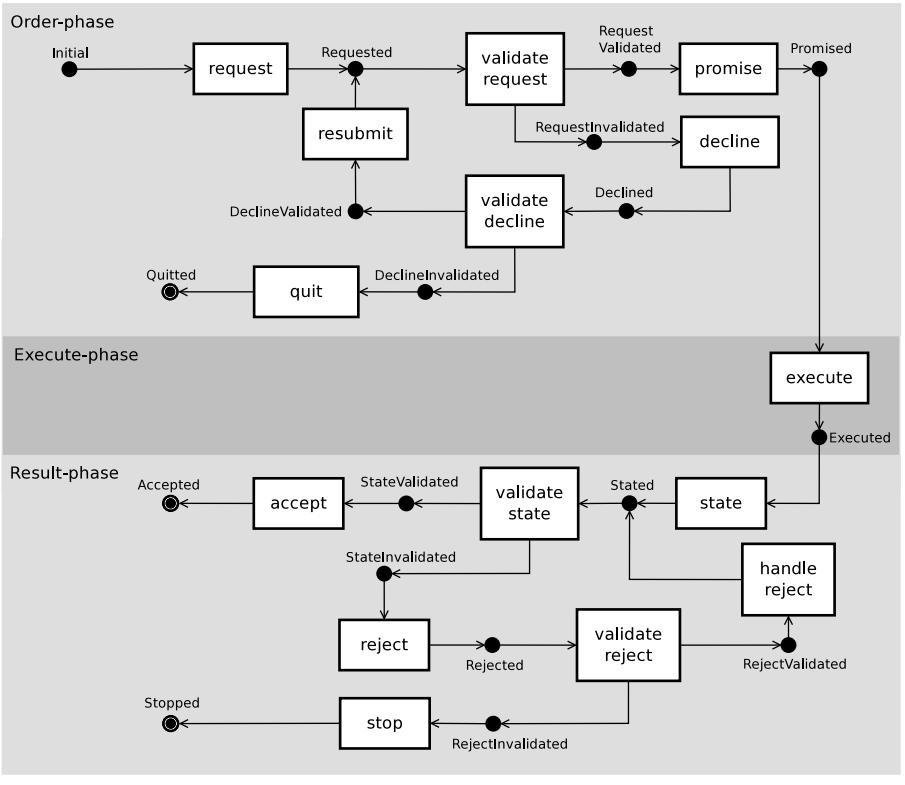

Fig. 3. Graphical representation of the standard transaction pattern flow

Table 3. Specification of the cancelation pattern flow element

Workflow name		Transaction Cancelation	
Data element		T01-CancelRequest	
Start state	**Action element**	**End state**	**Failed state**
Initial	CheckValidity	CancelRequestValid	
CancelRequestValid	AcceptCancelation	Allowed	not-Allowed
not-Allowed	Refuse	Refused	
Allowed	CancelTransaction	Canceled	

If the cancelation is allowed, it may be necessary to partly or completely roll back the transaction. Given the divergence of business contexts, a roll back can imply different actions given the state of the transactions. Therefore, the cancelation process will be designed using multiple scenarios implemented as separate action elements on the same life cycle data element. Consider the case where various parts are ordered to complete the assembly of a product. In case the parts have not yet been received, an order cancelation can be submitted

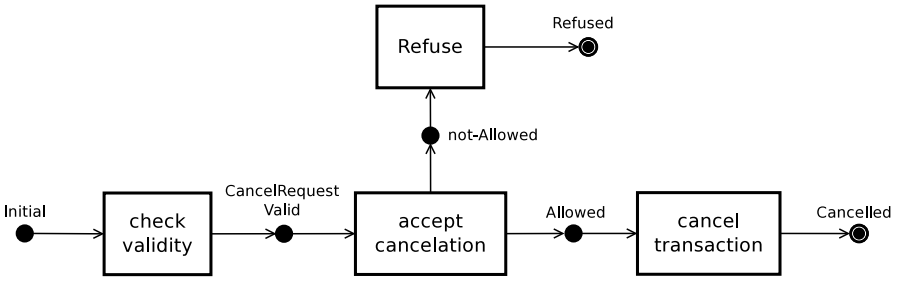

Fig. 4. Graphical representation of the cancelation pattern

to the parts supplier. In case the parts are already received and reserved, they should be released and made available for future assemblies. Thus, the scenario and constituent action elements are dependent on the life cycle data element's state when the cancelation request is initiated.

Since a cancelation can occur regardless of the current state of the transaction, it is modelled in the Enterprise Ontology transaction pattern as a separate entry point. However, the Normalized Systems theorems do not allow that the state of the main flow is simply altered by any other flow because a flow element actively interfering with another flow element is considered a so-called *GOTO statement*. In accordance with the seminal work of Dijkstra [3], Normalized Systems does not allow this kind of statements, and therefore prohibits such a direct state transition by another flow.

We outline the solution for adding cancelation patterns consistent with Normalized Systems theorems as described in [16]:

- A `cancelRequest` data attribute is added to the data element operating the flow.
- A cancel can be initiated in multiple ways. The particular situation should be assigned to the value of the `cancelRequest` data attribute by the `CancelTransaction` standard action element.
- The engine operating the respective flow element checks the `cancelRequest` data attribute. If this field is set, the current state of the flow will be saved in the so-called *parking state field*. The regular state field of the workflow will be set to "cancel requested".
- An action element will subsequently be triggered to decide which cancelation flow—i.e., sequence of action elements on the corresponding life cycle data element—has to be triggered as the cancelation scenario will differ according to the life cycle state as also illustrated by the cancelation patterns in Enterprise Ontology. Therefore, this action element will use the value of the so-called *parking state field*, uniquely describing the life cycle state of the corresponding data element when the cancel request was communicated.

This implies that a cancelation is handled as a sequence of action elements on the same life cycle data element. This is in line with the observation that requesting,

promising, executing, stating, declining, or cancelling a fact addresses the same concern. However, the sequence of actions about the cancel request itself are separated in their designated elements. It should be noted that we present a generic cancelation pattern. The possibility of triggering different cancelation flows, based on the value of the `cancelRequest` data attribute, allows us to implement the four different Enterprise Ontology cancelation patterns.

4 Application in Enterprise Architecture

In the previous section, we presented a translation of the Enterprise Ontology transaction pattern in Normalized Systems constructs. This artefact could be used in the context of enterprise architectures. We now outline the implication of our artefact in enterprise architectures as defined by Hoogervorst [7]. Hoogervorst proposes a method to design so-called construction models that enable the implementation of the implementation-independent Enterprise Ontology models. Based on the ontological models, four enterprise design domains (i.e., business, organization, information and technology) need to be designed. Enterprise architecture provides "the normative guidance for the design process" [7]. The architecture consists of principles, which have to be respected during the design of the construction models. These principles are the result of strategic choices. Therefore, organizations with identical ontological models can be implemented differently based on their different architectural principles, since different construction models will be designed. This is how organizations can differentiate from each other.

However, certain characteristics can be useful for any organization, such as evolvability. When the architecture needs to achieve such general strategic characteristics, architectural principles could be proposed which are more generally accepted. To achieve this, we need to know which principles affect the evolvability of construction models. According to Normalized Systems, the occurance of combinatorial effects affects evolvability. Principles which are analogous to the Normalized Systems theorems could thus affect the occurance of combinatorial effects—and therefore, evolvability—in contruction models. Such principles would need to guide the implementation of transactions to avoid combinatorial effects. Therefore, our implementation of the transaction pattern seems to fit the concept of enterprise architecture as intented by Hoogervorst: it guides the design of the transaction implementation by restricting design freedom, since only Normalized Systems elements can be used. Our artefact provides a basis for the further development of construction models which are free of combinatorial effects. The use of our construct is not limited to the design domain *information technology*. For example, designing the processes of the design domain *organization* based on our artefact enforces adherence to the Normalized Systems theorems, while respecting the integration between the processes which implement a certain transaction.

5 Discussion and Conclusions

This paper presents the first implementation of Enterprise Ontology transactions with explicit attention to combinatorial effects. It has two important contributions. First, our artefact shows that a mapping between Enterprise Ontology and Normalized Systems constructs is feasible. More specifically, it shows that such a mapping is feasible very early in the design process. Moreover, we presented a generic and systematic mapping. While it is possible that the mapping artefact needs to be refined or adapted, it can be used for the implementation of any transaction. This means that our artefact can be used as a starting point for designing evolvable organizations. We further illustrated this point by suggesting the use of the artefact within enterprise architectures. While our implementation remains at an abstract level, further specification of construction models can be guided by existing research, both scientific and practical. On the scientific level, Normalized Systems has proven to prevent combinatorial effects in software implementations. On the practical level, large-scale mission-critical systems are already developed using Normalized Systems elements.

Second, our mapping shows that the Normalized Systems theorems do impact the implementation of Enterprise Ontology models, when combinatorial effects need to be avoided. In order to implement transactions which are free of combinatorial effects, several guidelines can be prescribed:

- Additional state transitions need to be created in order to comply with the separation of concerns and separation of state theorems. We introduced these state transitions during the mapping of the standard transaction pattern.
- Based on previous research, we propose an implementation of the cancelation patterns which enables an implementation of different roll-back scenarios and adhering to the Normalized Systems theorems.

Following these guidelines will not affect the Enterprise Ontology models itself, since they are implementation-independent. The occurrence of combinatorial effects during changes will only affect the actual implementation of the Enterprise Ontology models. While we do not claim to have removed all combinatorial effects in our implementation, we achieved an effective and efficient mapping method by specifying these guidelines early in the design process. Effective, because the use of Normalized Systems elements implies the adherence to architectural principles. Efficient, because combinatorial effects are prevented instead of removed.

The presented approach suggests following future research subjects. First, we presented the mapping of a single transaction. Obviously, the construction of an organization implies the integration of several transactions. In subsequent research, we will focus on an approach to integrate different transactions, while respecting the Normalized Systems theorems. While the current mapping is mainly influenced by the separation of concerns and separation of states theorems, it can be expected that guidelines for integration will need to focus on the data and action version transparency theorems. Second, the focus of this paper was on the

conceptual mapping of constructs. In following publications, we will report on the applications of our artefact in various cases.

References

1. Auramäki, E., Hirschheim, R., Lyytinen, K.: Modelling offices through discourse analysis: the sampo approach. Computer Journal 35(4), 342–352 (1992)
2. Dietz, J.L.: Enterprise Ontology: Theory and Methodology. Springer, Berlin (2006)
3. Dijkstra, E.: Go to statement considered harmful. Communications of the ACM 11(3), 147–148 (1968)
4. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J., Mockus, A.: Does code decay? assessing the evidence from change management data. IEEE Transactions on Software Engineering 27(1), 1–12 (2001)
5. Goldkuhl, G.: Generic business frameworks and action modeling. In: Proceedings of the Conference on Communication Modeling—Language/Action Perspective 1996, Springer, Heidelberg (1996)
6. Hammer, M.: Reengineering work: Don't automate, obliterate. Harvard Business Review 68(4), 104 (1990)
7. Hoogervorst, J.A.P.: Enterprise Governance and Enterprise Engineering (The Enterprise Engineering Series), 1st edn. Springer, Heidelberg (2009)
8. Lehman, M.: Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 68, 1060–1076 (1980)
9. Lehman, M.M., Ramil, J.F.: Rules and tools for software evolution planning and management. Annals of Software Engineering 11(1), 15–44 (2001)
10. Leist, S., Zellner, G.: Evaluation of current architecture frameworks. In: SAC 2006: Proceedings of the 2006 ACM symposium on Applied computing, pp. 1546–1553. ACM, New York (2006), http://doi.acm.org/10.1145/1141277.1141635
11. Mannaert, H., Verelst, J.: Normalized Systems—Re-creating Information Technology Based on Laws for Software Evolvability, Koppa, Kermt, Belgium (2009)
12. Mannaert, H., Verelst, J., Ven, K.: Exploring the concept of systems theoretic stability as a starting point for a unified theory on software engineering. In: Mannaert, H., Ohta, T., Dini, C., Pellerin, R. (eds.) Proceedings of Third International Conference on Software Engineering Advances (ICSEA 2008), pp. 360–366. IEEE Computer Society, Los Alamitos (2008)
13. Mulder, H.: Rapid enterprise design. Ph.D. thesis, TU Delft (2006)
14. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
15. van Reijswoud, V.: The structure of business communication: Theory, model and application. Ph.D. thesis, Technische Universiteit Delft (1996)
16. Van Nuffel, D., Mannaert, H., De Backer, C., Verelst, J.: Deriving normalized systems elements from business process models. In: International Conference on Software Engineering Advances, pp. 27–32 (2009), http://doi.ieeecomputersociety.org/10.1109/ICSEA.2009.13
17. Winograd, T., Flores, F.: Understanding Computers and Cognition: A New Foundation for Design. Addison Wesley, Reading (1986)