Baltic J. Modern Computing, Vol. 2 (2014), No. 3, 132-149

On Applying Normalized Systems Theory to the Business Architectures of Information Systems

Erki EESSAAR

Department of Informatics, Tallinn University of Technology,
Akadeemia tee 15A, 12618 Tallinn, Estonia

Erki.Eessaar@ttu.ee

Abstract: Application of the database normalization theory and the principle of orthogonal design allows us to reduce data redundancy in relational/SQL databases and hence avoid certain update anomalies and simplify development of database applications. The normalized systems theory for achieving modular and highly evolvable systems has similar goals in case of distributing the tasks of a system between its elements. We have suggested data-centric practices for finding the functional and data-centric subsystems of transactional information systems by identifying and analyzing the main business entity types and their life cycles. These subsystems and their interconnections constitute a large part of information systems' business architectures. The specification of business architecture helps us to decompose the system development problem into more manageable subproblems. This paper studies whether the practices contribute towards creating normalized systems. We present a metamodel of the subsystems-related elements of data-centric practices and a metamodel of elements of normalized systems.

Keywords: information system, business architecture, subsystems, normalized system, modular structure, combinatorial effects, entity type, evolvability.

1. Introduction

The business architecture of an information system (IS) describes its layered decomposition. It does this by specifying different types of subsystems (areas of competence, functional subsystems, and data-centric subsystems – registers) and their interconnections (Roost et al., 2001; Roost et al., 2004). It is a good basis for the development of the system in a piecemeal fashion and planning its entire development process. There could be many alternative decompositions of an information system. In (Eessaar, 2014b), we proposed data-centric practices for the specification of business architectures of transactional information systems based on the identification and analysis of the main business entity types (*main entity types* in short) and their life cycles. Based on these, we can quickly identify functional subsystems and registers as well as their interconnections that form a big part of the information systems' business architectures.

The business architectures should have high quality. How to evaluate the quality? For instance, one could use the model of good decomposition (Weber, 1997) [referred by Burton-Jones and Meso (2002)], which specifies five conditions required for a good decomposition. Although very useful, these criteria do not explicitly consider

evolvability that is an important characteristic of systems in our ever-changing world. One could use patterns of assigning responsibilities to classes and objects (General Responsibility Assignment Software Patterns, GRASP) (Larman, 2002) for the evaluation. It raises a question what of these patterns would be applicable in case of subsystems.

We think that that the emerging normalized systems (NS) theory (Mannaert et al., 2011) provides a suitable basis for evaluating the quality of information systems' business architectures. The main goal of the paper is to study, whether the application of the data-centric practices produces business architectures that one can characterize as NS. NS theory is suitable for the task because it deals with investigating the behavior of evolving modular systems and on the other hand, one can use the data-centric practices to specify the modular structure of a certain type of information systems. To achieve the goal, we need a better understanding of NS theory. Hence, a subgoal of the paper is the creation of a metamodel of NS theory elements. Another subgoal is to demonstrate that it is possible to discuss business architectures in terms of the elements of NS. To make this kind of discussion possible, there must be a mapping between the elements of the two approaches. Therefore, we also have to present a metamodel of the elements of data-centric practices that are related to subsystems to facilitate understanding of the practices and the mapping.

We organize the rest of the paper in the following way. First, we shortly explain the main principles and elements of both NS theory and relevant data-centric practices for specifying business architectures. In addition, we present their metamodels based on our current understanding of their elements. These are also contributions of the paper. We point that the practices represent a development approach of information systems that is again gaining attention. We refer to some related works about identifying and characterizing subsystems. Secondly, we discuss the quality of the resulting business architectures in terms of NS theory. Finally, we conclude and point to the future work with the topic.

2. Information Systems' Business Architectures

The idea of using the main entity types (also known according to Sanz (2011) as significant entities, principal entities, subjects of process, real-world entities, essential business entities, and artifacts) and their life cycles as the basis of modeling processes is not new (Sanz, 2011; Ould, 1997). Recently, the interest towards entity-centric development approaches of information systems in general and business processes in particular has started to grow again due to the artifact-centric process management paradigm (Dumas, 2011). In (Eessaar, 2014b), we proposed data-centric practices for the development of transactional information systems that are inspired by this approach and by a methodological framework for Enterprise Information System (EIS) strategic analysis and development (Roost et al., 2001; Roost et al., 2004). Fig. 1, Fig. 2, and Fig. 3 present a partial metamodel of the practices, concerning mainly decomposition of information systems into subsystems. Classes with the grey and white background depict business and information system concepts, respectively.

The idea of decomposing information and software systems into subsystems to facilitate better comprehension, easier learning, development, maintenance, reuse, and evolution of the systems is of course not new. Parnas (1972) names advantages of the decomposition and advises how to reach to a good program decomposition. Bergland

(1981) reviews four program structuring methodologies. Wand and Weber (1990) formalize concepts *subsystem* and *decomposition* of information system. Wand and Weber (1990), Wand and Weber (1995), and Weber (1997) discuss the characteristics of good decomposition. However, both Bergland (1981) as well as Wand and Weber (1990) write about functional decomposition whereas framework for EIS strategic analysis and development and the data-centric practices foresee *three* different types of subsystems to separate concerns – areas of competence, functional subsystems, and registers. This distinction is also not new. For instance, these subsystem types correspond bijectively to the three aspects of the ArchiMate language for enterprise architecture (Berrisford and Lankhorst, 2009) – active structure, behavior, and passive structure, respectively.

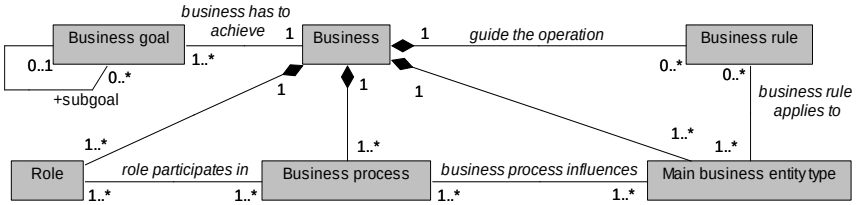

Fig. 1. A part of the metamodel of the elements of data-centric practices

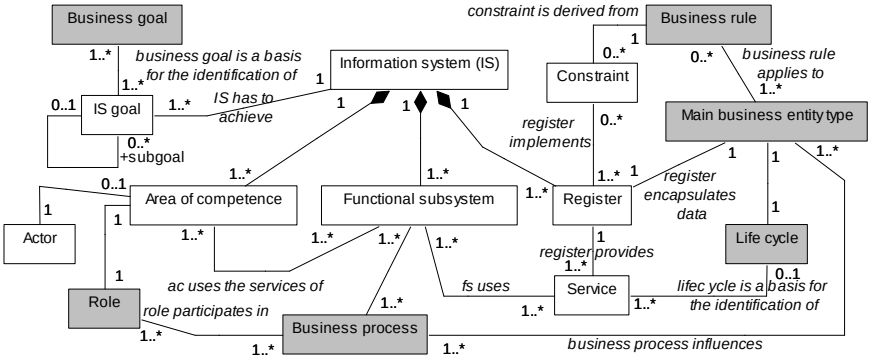

Fig. 2. A part of the metamodel of the elements of data-centric practices

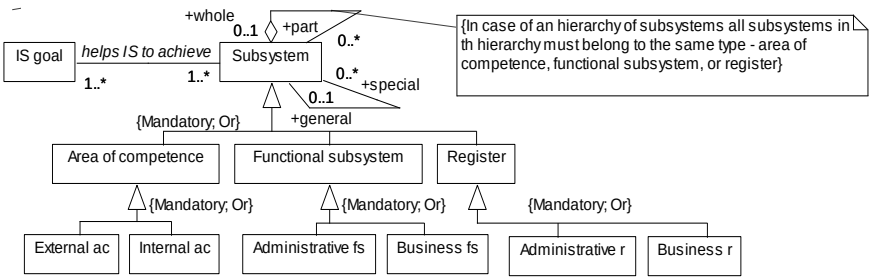

Fig. 3. A part of the metamodel of the elements of data-centric practices

Berrisford and Lankhorst (2009) note that the aspects have been derived from the deep grammatical structure of natural languages in which sentences consist of subjects, verbs, and objects.

Wand and Weber (1995) explain concepts *surface structure* and *deep structure* of information systems. By specifying the areas of competence of an information system, we describe its *surface structure* that determines how users see the system – what services (functionality) the information system should provide to different areas of competence to support their goals and what characteristics values the product and its user experience (see ISO/IEC 25010:2011) should have. It is possible that representatives of areas of competence themselves model how the system should appear to them (Roost et al., 2001).

Functional subsystems and registers (data-centric subsystem) are elements of the *deep structure* of information systems and therefore, according to Wand and Weber (1995), manifest the *meaning* of the real-world system that the information system intends to support. They do it by reflecting processes and state of the real-world system, respectively. Elements of surface structure change frequently because human's view of the world evolves and the views that information systems offer to humans must evolve accordingly. On the other hand, deep structure elements – functional subsystems and registers – are more stable and their meaning does not change, for instance, if the organizational structure or implementation technologies change. Wand and Weber (1995) note that "good deep structures provide inherent stability to information systems in the face of change" (p. 206). It shows, why the investigation of the goodness of the result of breaking the system up into subsystems is important.

Sanchez and Mahoney (1996) point that modular design is possible in case of many different types of products, not only software systems. The idea of finding and using subsystems in general and the business architectures in particular is the manifestations of the *divide and rule* approach that is the most common way of dealing with the complexity (Green and Leisman, 2011). By finding the business architecture of an information system and creating it based on the architecture, we reduce the complexity of the system by breaking it into encapsulated units (subsystems) that are more manageable. By doing this, we reduce the number of connections within the system and modularize the system. Ideally, we would like to see that subsystems as modules have high internal connectivity and relatively low connectivity with other modules (subsystems) (Green and Leisman, 2011). Low connectivity (loose coupling) between subsystems allows us to develop the subsystems and change their internals relatively independently from other subsystems. This kind of architecture also supports extension of the system in terms of adding relatively easily new subsystems and reusing existing models and implementations of subsystems to extend the system. It has also effect to the structure of development organization. According to Sanchez and Mahoney (1996) loosely coupled modules that have standardized interfaces make possible loosely coupled product development organization where development takes place "autonomously and concurrently under the embedded coordination of a modular product architecture" (p. 68). Business architecture is an example of modular product architecture in the domain of information systems.

The subsystems that we propose to find communicate with each other and form a layered communication network where each level consist of a different type of subsystems. Subsystems from one level invoke the services offered by the subsystems from the next lower level. Each service is accessible through an interface (Berrisford and Lankhorst, 2009). Areas of competence use functional subsystems through a user

interface. Functional subsystems use the data management services (database operations) provided by registers. These services have interface. Implementation of these services forms the Virtual Data Layer of the database (Burns, 2011). To summarize, areas of competence use data in registers with the mediation of functional subsystems.

According to the data-centric practices, one has to identify the main entity types. Each of these will have a corresponding separate register and generally also a functional subsystem in the business architecture (the only exception is related to the management of state change events – see Sect. 4.1). The latter provides functionality to the users for managing data corresponding to the main entity type and its related (not main) entity types through its entire life cycle. In each such pair of subsystems, the functional subsystem manages data in the register (creates, reads, possibly updates and deletes data in it). Each functional subsystem has to read data from zero or more other registers to get data that enables its functionality. We suggest using the state-transition (life cycle) models of the main entity types for finding interactions between the subsystems, use cases that specify functional requirements to the functional subsystems, and services that the registers have to offer to functional subsystems to encapsulate data structures and data values of registers. Each use case should describe an elementary business process that helps some area of competence to achieve its goals.

In general, for each state transition of a main entity type there will be a separate use case of the functional subsystem and a separate service offered by the register for recording the new state. It means that we will have reactions to the external or internal events of the system that trigger the state transitions. The interactions between the life cycles of different entity types give us information about what functional subsystems manage data in each register. The idea of using the main entity types is, for instance, similar to the Riva development method (Ould, 1997). Its essential business entities and case management processes correspond well to the main entity types and functional subsystems, in the data-centric practices. Hence, the approach of this paper should be interesting for the evaluators of the entity-centric development approaches in general.

The approach of finding functional subsystems offered by the data-centric practices has the same basic premise than the functional decomposition methodology that is based on data structure design (Bergland, 1981) that "a correct model of the data structures can be transformed into a program that incorporates a correct model of the world" (p. 26). Bergland (1981) refers to different qualitative levels of *cohesion* that is the "glue" that holds each module (including subsystem) together. In our view, three highest levels of cohesion characterize the functional subsystems and registers that one identifies by using the data-centric practices. In case of functional subsystems and registers there must be cohesion of services, offered to the areas of competence and functional subsystems, respectively. Each such subsystem has *communicational cohesion* because it operates on common data grouped together to the same registers. Each such subsystem has *sequential cohesion* because its offered services correspond to the sequential steps of the life cycle of the same main entity type. Finally, each such subsystem has *functional cohesion* (the highest level of cohesion according to Bergland (1981)) because all its submodules contribute to performing one single function – manage data of a main entity type and its related (not main) entity types through its entire life cycle.

Green and Leisman (2011) differentiate between whole-part and general-special relationships between modules. Same types of relationships between the subsystems are possible. Subsystems participating in such relationships must have the same type. One can indicate general-special relationship between subsystems (see Fig. 3) for complexity management purposes. Let us consider the data-modeling pattern *Party* (Fowler, 1997)

as the basis of an example. Piho (2011) notes that *Party* is an archetype pattern for modeling of an independent phenomenon of enterprises. Many information systems have to keep track of persons and locations (organizations and organization structure) and hence benefit from the use of *Party* pattern in the modeling process. In this pattern, *Party* is the supertype and *Person* as well as *Organization* are its subtypes. Next, we list possible combinations of registers that one could suggest in case of an information system that has to keep track of parties. According to the data-centric practices, there would be exactly the same functional subsystems to manage data in the registers.

- Only *register of parties* that also contains data about persons and organizations.
- Only *register of persons* and *register of organizations*. It means that we actually do not follow the *Party* pattern because there is no separate register that contains data common to all the parties (for instance, different types of addresses). The corresponding data structures are duplicated in the registers.
- *Register of parties* that contains data common to all the parties as well as *register of persons* and *register of organizations* for recording subtype-specific data. In this case, one could define general-special relationships between *register of parties* and registers that correspond to the subtypes of *Party* that also means inheritance between the subsystems. We see a benefit of indicating such relationships between subsystems in reuse of specifications that have been created in case of the more general subsystem and do not have to be repeated in case of more special subsystems due to inheritance. Semantics of inheritance in case of subsystems needs further elaboration and in our everyday practice, we have not indicated this type of relationships. The subtypes of *Party* could themselves have subtypes (for instance, *Worker* as a subtype of *Person*). In the context of business architectures it could mean multi-level inheritance hierarchy of subsystems.

Specifying containment (whole-part) relationships between the subsystems is also possible. However, in practice, we have not done that to avoid overly complicated specifications of subsystems.

3. Normalized Systems Theory

Information systems have to evolve over time because of changing business requirements, changing laws and regulations that influence the business as well as evolving technical platforms. Making changes in a system is not easy if its internal elements have low cohesion and high coupling and hence changes in one element cause many more changes in other elements. NS theory (Mannaert et al., 2011; De Bruyn and Mannaert, 2012; Verelst et al. 2013) is based on the observation that the presence of combinatorial effects influences the flexibility and evolvability of a system in a negative manner. The idea is that complexity of a change in a system should depend on the nature of the change itself and not from the size of the system. If there are combinatorial effects present in the system, then it means that the larger is the system, the more difficult it is to change it because the more changes has to be made in other elements of the system as the ripple effect. Hence, creating a system that conforms to NS theory should improve its maintainability. The theory formalizes knowledge about good program design (Bergland, 1981) that should result in a set of nested modules that one can connect in a hierarchy to form large programs.

An example of similarities between NS theory and the database normalization theory (Date, 2009) is that if there is a not fully normalized (not at least in the fifth normal form) base table (table in short), then the existence of certain intra-table dependencies between its columns leads to data redundancy and hence update anomalies. In this case, the more rows there are in the table, the more work is required in case of data updating operations (see Fig. 4). Therefore, the amount of work depends on the number of rows in the table. Failing to make all the required updates leads to inconsistent data.

In Fig. 5, we present tables *Car* and *Manufacturer* that are in the fifth normal form. We have eliminated the data redundancy of manufacturer names within the table *Cars_and_manufacturers* by replacing the original table with its decomposition into two of its projections. By doing this, we placed columns participating in the functional dependency *Manufacturer_ID* → *manufacturer_name* into the separate table *Manufacturer*. Now, if one wants to change the name of a manufacturer, then one has to update only one row in table *Manufacturer*. Similarly, in Fig. 4, we illustrate implementation of the same functionality (*Task 1*) in multiple times within the same module. Changing the functionality requires extra effort. Failing to make all the required changes leads to the inconsistent behavior of the system. To avoid that, one should decompose the functionality into a separate module that is accessible to other modules through an interface (see Fig. 5). In this case, the amount of work in updating the functionality (*Task 1*) that does not change the interface does not depend on the size of the system. Fig. 5 illustrates how submodular tasks have become actions at the modular level to separate concerns. It conforms to a prescriptive design theorem of NS theory.

The application of both theories has a goal to avoid update anomalies (of data and functionality, respectively) and produce a good representation of the real world. Both require nonloss-decomposition (of tables and modules, respectively) in a multi-step manner, and increase the number of tables and modules, respectively. An example of differences is that application of the database normalization process will not eliminate all the possible data redundancies within databases but the goal of NS theory is to create systems with no combinatorial effects.

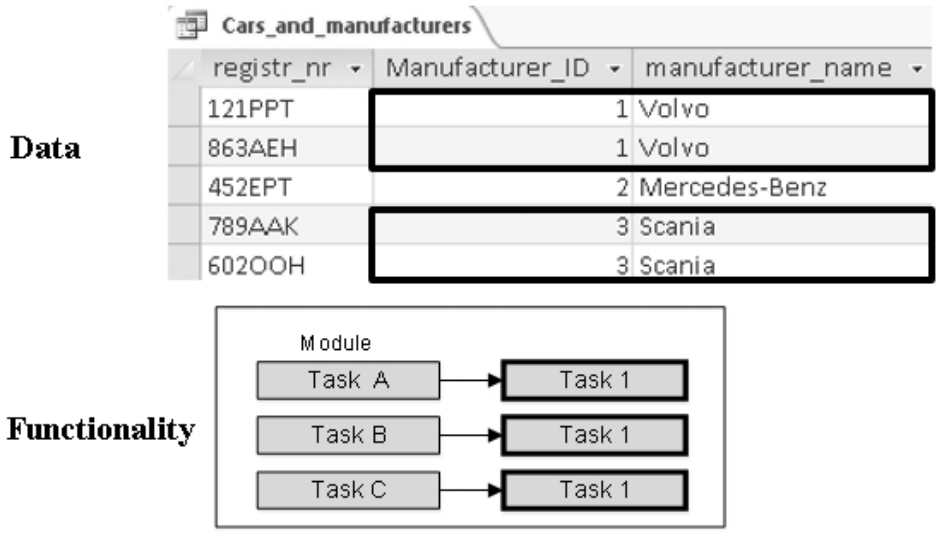

Fig. 4. An example of redundancy of data and functionality

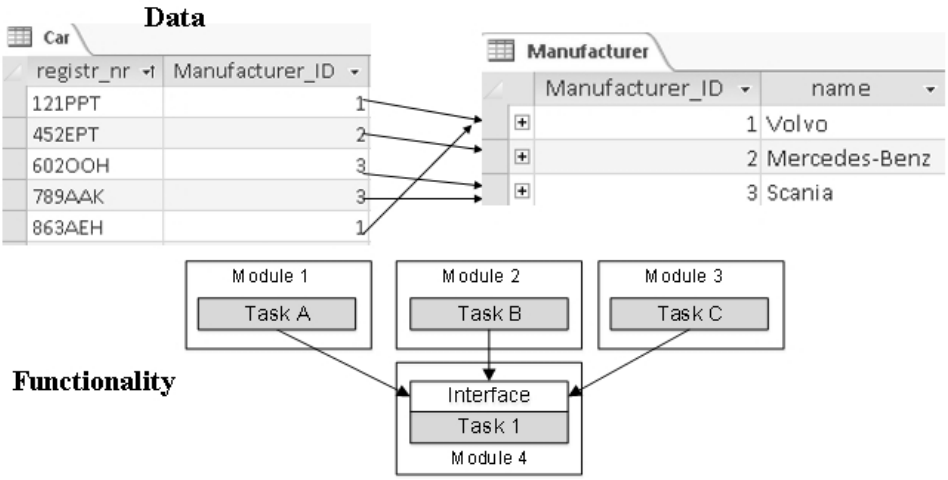

Fig. 5. An example of reduced redundancy of data and functionality

Another difference is that the normalization theory helps us to reduce data redundancy within a table but does not deal with the situations where data redundancy exists across multiple tables. In this case applies another formally defined principle called The Principle of Orthogonal Design (Date, 2009). On the other hand, the theorem about separation of concerns looks the system under investigation as a whole, from very coarse grained to very fine grained modules, and if some task is repeated within modules or in multiple modules, suggests decomposing it into a separate module.

NS theory defines constructs (primitives) that constitute such systems (see Fig. 6).

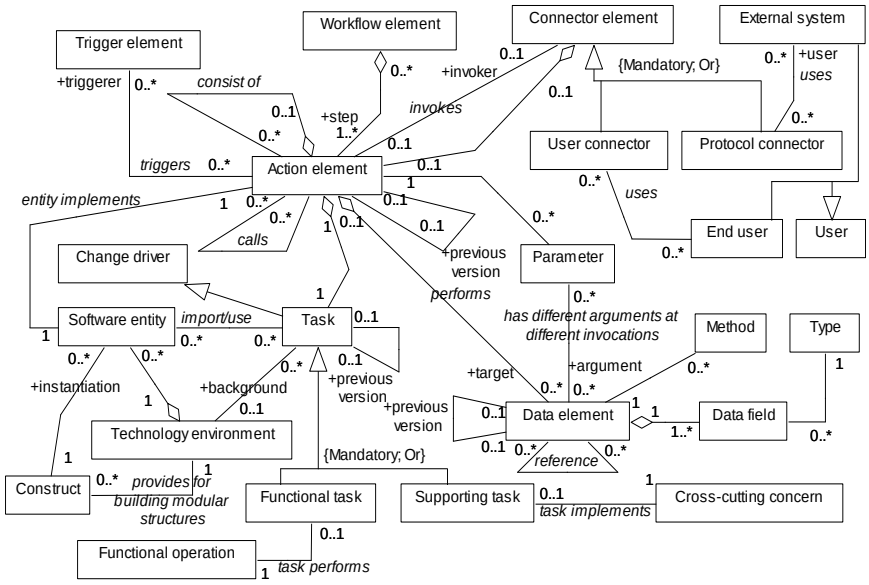

Fig. 6. A metamodel of the elements of NS theory

The constructs are general (data element, action element etc.), are not associated with specific software languages, platforms or paradigms, and represent *roles* that elements of systems from many different domains could have. For instance, there are examples of the use of NS theory in case of software systems (Mannaert et al., 2011), specifications of requirements (Verelst et al., 2013), and organizational structures (De Bruyn and Mannaert, 2012). Verelst et al. (2013) present examples of combinatorial effects that exist in the real world, outside the scope of computerized systems.

Description of different types of systems in terms of generic elements makes it possible to better understand deep similarities of systems or their parts. The similarities may not be apparent at first glance because the systems belong to different domains or the parts have different granularity (for instance, subsystems vs. classes).

A problem of the description of NS theory is the lack of clear distinction of types, values, and variables. For instance, Date (2006, 2009) clearly separates and defines these concepts. Therefore, it is difficult to understand whether *data entity* and *data element* are the same things (in the paper we assume they are) and whether they mean *type*, *value*, or *variable*. Another core concept is *operator* (Date, 2009) and to our understanding, this corresponds to action element.

Each NS has to conform to four prescriptive design theorems that ensures the lack of combinatorial effects. Next, we list the informal descriptions of the theorems (Mannaert et al., 2011; De Bruyn and Mannaert, 2012).

- *Separation of Concerns*, meaning that each change driver (external technology, task – they can evolve) in a system must be separated into a separate module. Fig. 5 illustrates implementation of each change driver (task) in a separate module.
- *Data Version Transparency*, meaning that there could be multiple versions of data elements without affecting action elements producing or consuming these.
- *Action Version Transparency*, meaning that it must be possible to modify action elements without affecting their calling elements.
- *Separation of States*, meaning that system has to keep state after every action that belongs to a workflow.

NS theory describes a set of anticipated changes in terms of the elements. One has to translate each change in the business requirements to a set of these changes. Next, we explain the elements of business architectures in terms of the elements of NS.

4. A Discussion of Business Architectures of Information Systems in Terms of NS Theory

Areas of competence can be external or internal to the organization and their modeling defines different views to the functionality and data (action and data elements) of the system that reflect different goals of the users of the system. In Eessaar (2014a), we claimed that concept *Area of competence* (see Fig. 3) has only corresponding concept *User* in NS metamodel. After more consideration, we think that concept *Area of competence* (see Fig. 3) has corresponding concepts *User*, *Connector element*, *User connector element*, and *Protocol connector element* in NS metamodel (see Fig. 6). Representatives of areas of competence are users of the system. However, in case of each area of competence (as a subsystem), someone (representatives of the areas of competence, developers, or their combination) has to describe how the system should appear to users that belong to it. In terms of NS elements it means that one must describe

a new *Connector element*. If the representatives of an area of competence would use the system through a user interface that would be a part of this system, then in terms of NS theory one has to define a *User connector element*. If the representatives of an area of competence need from the present system some services through an external application, then in terms of NS theory one has to define a *Protocol connector element*.

If we want to show in case of using the data-centric practices that the system periodically triggers a use case, then we associate it with the special actor called *Time*. In case of NS theory there is a *Trigger element* that can trigger *Action elements*. In terms of specifying areas of competence, one can either define area of competence *Time* or associate the use case with a human area of competence that really has the responsibility to achieve the goals of the process and who would have to perform the process if there is no automation (Crain, 2002).

Concept *Functional subsystem* (see Fig. 3) has corresponding concepts *Workflow element* and *Action element* in NS metamodel (see Fig. 6). Mannaert et al. (2011) note that concept *workflow element* is based on the concept of a state machine. Similarly, in case of using the data-centric practices, each functional subsystem is a workflow that corresponds to the life cycle of the main entity type based on that one proposed the subsystem. For example, *order management functional subsystem* manages orders through their entire life cycle (see Fig. 7). Each functional subsystem corresponds to a main function of a system, and hence we think about them as coarse-grained action elements. Each workflow element performs a number of action elements on a specific target life cycle data element that is used to keep state (Mannaert et al., 2011). Each such action element corresponds to one use case (see, for instance, action element *Submit order* in Fig. 7). The life cycles of the main entity types determine the order of invoking the action elements. Successful execution of these action elements may change the state of a main entity and that will be captured in the system (in a register). It makes possible new state changes of the entity and hence new executions of action elements.

A business processes could involve multiple business entity types and roles. One can model the workflows that span multiple functional subsystems, registers, and areas of competence by analyzing interactions between the life cycles of different main entity types where a state transition in one causes a state transition in another.

Although the main task of registers is to record data, we cannot characterize registers as only data elements (entities) because "data entity only contains data (as in a structure or record) and does not have an interface" (Mannaert et al., 2011, p. 94). At the same time Mannaert et al. (2011) write that data elements have methods to enable data-version transparency. In our view methods are action elements that implement interface for accessing internal variables of objects that contain values. Hence, we find this description of data entities confusing.

Each register does have an interface that is the union of the interfaces of its services. The quote from (Mannaert et al., 2011) in the previous paragraph describes data elements as variables (containers) that contain values. Similarly, we can describe the union of data structures of a register as a variable. By rewording the definition of *database variable* (Date, 2006) a *register variable* is a variable whose value at any given time is a register value. The type and hence the structure of a variable can evolve and it may have different values at different times. NS theory speaks about anticipated changes in the information systems and one of the changes is the creation of an additional data element (Mannaert et al., 2011). Here the talk is about variables. However, almost right after this sentence the paper states that another anticipated change is the creation of "an additional action entity having a specific data entity as input, or producing a specific data

entity as output" (Mannaert et al., 2011, p. 95). This describes data elements as types or values. Mannaert et al. (2011) also write "arguments and parameters must be encapsulated data entities" (p. 102). This is a confusing description because there is a logical difference between parameters and arguments (Date, 2009). Each parameter has a type (named set of values (Date, 2009)) that determines possible arguments that can correspond to the parameter. Arguments are actual operands that replaces parameters in invocations of operators (Date, 2006). Each argument must be of the same type as the parameter. Each argument is either a variable (if and only if the corresponding parameter is subject to update) or a value (Date, 2006). Mannaert et al. (2011) also write that data elements have methods. It also suggests that data elements are types or values. In this context, it is difficult to understand whether a data element (data entity) as described by NS theory is a *variable*, a *type*, or a *value*.

Concept *Register* (see Fig. 3) has corresponding concepts *Action element* and *Data element* in NS metamodel (see Fig. 6). Each register is an action element that consist of other action elements that correspond to the services offered by the register (see Fig. 2). The services perform data management actions that encapsulate a data element that is also a part of each register. *Data fields* (see Fig. 6) correspond to individual data structures (for instance, base tables in case of SQL databases) in the database. The encapsulated data element is in this case the union of data structures for recording data that corresponds to a main entity type and its related (not main) entity types. Each service creates, reads, updates, or deletes data in one or more data structures. In case of registers, the background technology would usually be a database management system (DBMS). If one uses a SQL DBMS to implement registers, then one can implement action elements that correspond to services as procedures, functions, views, and materialized views (snapshots) in the background technology. These database objects are *software entities* that are created by instantiating the *constructs* of DBMS *technology environment* (see Fig. 6).

Fig. 7 illustrates mappings between the elements of NS theory and some elements of the data-centric practices by using an example. References to NS elements are denoted with bold font. For the presentation purposes, we simplified the life cycle of orders.

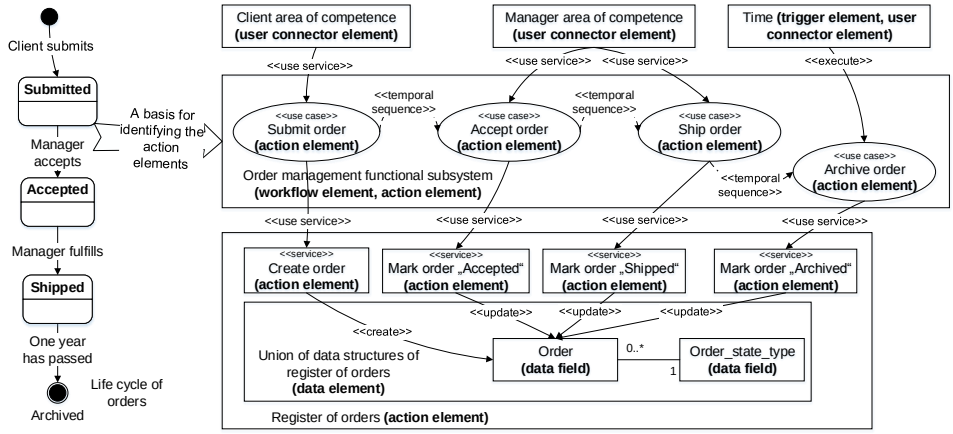

Fig. 7. An example of description of IS business architecture in terms of NS elements

4.1. Separation of Concerns

Each action element can contain only a single task according to the theorem about separation of concerns (Mannaert et al., 2011). System evolution could cause the change of each such task and therefore tasks are change drivers. Separation of concerns means that one must separate each change driver into a separate action element. Fig 4. and Fig. 5 illustrate changes in the structure of modules to achieve separation of concerns. Mannaert et al. (2011) define the theorem at the submodular level but acknowledges, "It is the designer's decision to which level tasks are considered" (p. 94). In this paper, we apply NS theory at the more coarse-grained level – to the elements of information systems' business architectures.

Mannaert et al. (2011) note that a practical manifestation of this theorem is the use of multi-tier architectures. It is also the result of using the methodological framework for EIS strategic analysis and development (Roost et al., 2001; Roost et al., 2004) in general and the data-centric practices (Eessaar, 2014b) in particular. The resulting business architectures have three tiers (areas of competence, functional subsystems, and registers), and each tier has its own task.

In our view, each functional subsystem and register is a coarse-grained action element. The use of data-centric practices results with the business architecture where each register has *the functional task* to offer data management services (create, read, possibly update and delete) related to a main entity type and its related (not main) entity types. Hence, for instance, *register of orders* does not contain data about goods (another main entity type) or services for managing data about goods. If it does, it would be a clear violation of the theorem. It contains references to goods that is a part of each order.

The use of data-centric practices results with the business architecture where each functional subsystem (as a workflow element that is also a coarse-grained action element) has *a functional task* to offer to the users services that are related to the management of data that corresponds to a main entity type and its related (not main) entity types based on the life cycle of the main entity type. However, the life cycles of different main entity types may interact, producing workflows spanning multiple functional subsystems and registers. For instance, if paying the invoice of an order and hence changing its state causes changing of the state of the order as well (invoice and order are some of the main entity types of the domain in question), then there is a dynamic relationship between the life cycles of *Invoice* and *Order*. *Invoice management functional subsystem* manages data about invoices in *register of invoices* but it also has to invoke a service offered by *register of orders* to register the new state of an order. Hence, the functional subsystem has multiple tasks that is not consistent with the theorem. If the life cycle of a main entity type does not interact in this way with the life cycles of other main entity types, then the corresponding functional subsystem has one task.

A problem of the description of NS theory (Mannaert et al., 2011) is that it does not clearly define concept *change driver*. If the life cycle of an entity type E or requirements about what data the system must capture in case of E evolve, then it causes changes in the functional subsystem and register corresponding to E. However, it could also lead to the changes in other functional subsystems and registers due to the interaction of life cycles of different entity types. For instance, adding possibility to cancel orders causes the need to do something with the corresponding invoices. It could be that users of *invoice management functional subsystem* want a report of such invoices. A new requirement to present certain information on printed invoices may cause the need to

collect the information during the creation of orders and register the information in *register of orders*. Hence, we see that a subsystem might have a single task but multiple change drivers.

We propose to have generally a separate use case/service for each state transition in the life cycles of the main entity types (see Fig. 7) in case of functional subsystems and registers, respectively. The result could be that these more fine-grained action elements contain only a single task and hence have only one change driver. For instance, we do not put the registration of all the possible state changes together into one use case or into one service of a register. However, a use case that has to change the state of main entities that have *different* types has multiple change drivers. Similarly, services that implement predefined queries that the system has to answer and need data from multiple registers have multiple change drivers.

Next, we give examples of combinatorial effects that the data-centric approach of finding subsystems helps to avoid. The data-centric practices distinguish business and administrative functional subsystems and registers (Eessaar, 2014b). Examples of administrative functional subsystems and registers are subsystems for managing data about invoices, contracts, documents, classifiers, workers, rooms, and buildings. Whether these subsystems are administrative or not depends on the nature of the organization that information system we specify. Fulfilling the administrative tasks supports accomplishing the goals of the organization but these tasks (and goals that correspond to these) are not the reason of its current operation. The tasks of administrative functional subsystems and registers are similar to the *supporting tasks* that require separate action elements (Mannaert et al., 2011). Combinatorial effects appear if one distributes administrative tasks among the business related functional subsystems and registers. For instance, if all or most of the registers contain data of some classifiers and their corresponding functional subsystems allow users to manage data of these classifiers, then changes in the general principles of classifier management (for instance, decision is made to start the recording of textual descriptions of classifier values) requires changes in many functional subsystems and registers. It is better to have classifier management task in a separate subsystem and hence the data-centric practices advise the specification of *classifier management functional subsystem* and *register of classifiers* in the business architectures. Detailed modeling of the *classifier management subsystem* will reveal separate use cases that describe management of different types of classifiers. These use cases are also action elements (see Fig. 7). This is also an implication of the separation of concerns theorem because "the more fine-grained the identification of the tasks by a designer, the more tasks are separated from each other" (Mannaert et al., 2011, p. 97).

The theorem about separation of concerns implies that implementation of cross-cutting concerns, such as logging history of changes, constitute separate tasks. These tasks must be implemented as separate action entities. If there is a need to keep track of the history of the state changes of entities that belong to a main entity type, then the data-centric practices require definition of an additional main entity type and specification of a separate register based on that. For instance, if there is a need to keep track of orders and also the history of state changes of orders, then the practices require specification of the main entity types *Order* and *Order event*. One has to specify *register of orders* and *register of order events* based on these main entity types. Order events that would be recorded in *register of order events* correspond to the state transitions in the model that describe the life cycle of orders (see Fig. 7). Functional subsystems that manage state changes of orders (mainly *order management functional subsystem* but

maybe also other subsystems like *invoice management functional subsystem*) have to invoke services of *register of orders* and *register of order events*.

The data-centric practices do not suggest general *register of events* for all the entity types because its data size would quickly become too large that reduces performance of the system. In addition, in case of different entity types there could be different requirements to the data that system has to capture in case of a state change event.

The creation of a separate register instead of recording this data in *register of orders* is a manifestation of the separation of concerns theorem, which requires that each action element can only contain a single task. Now *register of orders* contains services and data structures registering the current state of orders. *Register of order events* contains services and data structures for registering state change events. However, for instance, now *order management functional* subsystem as a high-level action element has to manage data that corresponds to multiple main entity types (*Order* and *Order event*). Hence, it has multiple tasks that violates the separation of concerns theorem. In addition, we have now multiple registers where similar (not the same) logging functionality and data structures are scattered. Because the registers of events implement a supporting task that is a cross-cutting concern, it would be right to characterize these as administrative registers.

We also note that according to the data-centric practices, one has to describe constraints to data as a part of specification of registers. In addition, the data-centric practices encourage enforcing constraints to data at the database level, as a part of implementing registers. The constraints should be implemented by using the means offered by the background technology (DBMS), preferably through a declarative manner. It ensures that any action element that accesses the data cannot violate the constraints. This is also a manifestation of the separation of concerns theorem. If we do not repeat descriptions and implementations of the constraints in case of other types of subsystems, then we have to do less work if we want to enforce new constraints, or modify/remove existing constraints.

4.2. Data Version Transparency and Action Version Transparency

The theorem about data version transparency writes about "Data entities that are received as input or produced as output by action entities" (Mannaert et al., 2011, p. 97). This statement describes data elements (data entities) as values. In the context of registers, we can speak about the value of a register that is held by the register variable. Invoking an action element (a service offered by a register) that helps system to encapsulate a data element (a part of the register) can change the value of the register variable.

The data-centric practices require that, in general, for each state transition of a main entity type there must be a separate service that is offered by the corresponding register to keep the state (see also section 4.3). Depending on requirements there could also be services that modify data about the main entities and their related entities without changing the state (according to the life cycle) of the main entities. In addition, there must be services that implement predefined queries that the system has to answer. These queries correspond to the informational needs of areas of competence. The queries might require data from multiple registers but for organizational purposes are assigned to one of the registers. The interface of a register is the union of the interfaces of its services.

The practices do not explicitly state that the interfaces of registers have to conform to the theorems about version transparency but it would be advantageous due to the

resulting improved evolvability of the system. Much depends on the background technology – usually DBMS in case of registers. If one implements registers as one or more SQL databases, then the services provided by registers would be implemented by using views, materialized views (snapshots), procedures, and functions. Further refinement of services may lead to the creation of new action elements that implement tasks that are needed by multiple services. For instance, to achieve separation of concerns one may implement fundamental stored procedures (FSP) (Burns, 2011) that are used by other more high-level procedures. Each FSP "performs one type of update (add, change, or delete) on one or more rows of data in a single database table" (Burns, 2011, p. 185).

Data version transparency means in the context of registers that it must be possible to make changes in the data structures of registers without affecting action elements that use these structures. In case of our approach, it is not entirely possible. Depending on the nature of the changes, one may have to change the internals of action elements that implement the services of the registers but does not have to change the action elements that use the services. Changes in the internals of data modification services are limited with the register that data structures have been modified. There could be multiple registers where one has to change internals of data reading services because registers may provide services that have to read data from multiple registers. However, there could also be changes of data structures that one cannot hide behind the services of registers. This kind of changes cause changes of the interface of the register and propagate to the action elements that use the services.

Action version transparency means that there could be multiple versions of action elements (services). One could use overloading of functions/procedures and use different schemas for different versions of database objects for achieving this state of affairs.

4.3. Separation of States

Invocation of use case action elements that constitute a functional subsystem corresponds to a stateful workflow that is a practical manifestation of the theorem. Action elements that correspond to use cases call action elements that correspond to the services provided by registers. Each calling is triggered by an external or internal event. Each successful calling of an action element that has the task to perform a data modification operation, results with state keeping in the register – it creates, updates, or deletes data about some entities and/or relationships. In addition, the registers contain explicit information about the current state of main entities with the help of state classifiers (see data fields *Order* and *Order_state_type* in Fig. 7 as an example). If a use case action element initiates a state transition that is consistent with the life cycle of the corresponding main entity type, then the action succeeds and data about the new state is captured in the register.

The practices do not require the use of asynchronous communication as suggested in (Mannaert et al., 2011). One should combine invocations of services into transactions that is a manifestation of the theorem (Mannaert et al., 2011).

5. Conclusions and Future Work

The paper focused on improving understanding of normalized systems (NS) theory and using it for the evaluation of business architectures that one has created with the help of

data-centric practices by identifying and analyzing the main entity types and their life cycles. The use of the practices leads us to the identification of functional subsystems and registers (coarse-grained modular structures). In the resulting business architecture each functional subsystem supports one type of entities going through their associated life cycle. It may change the state of other types of entities due to the interacting life cycles of entity types and capture the history of state changes. The current state (according to the life cycle) of the entities and other required data about the entities and their relationships are recorded in the corresponding registers. Both the data-centric practices and NS theory prescribe a modular structure of systems. The resulting business architectures violate in some places some of the prescriptive design theorems of NS theory. For instance, some of the functional subsystems have multiple tasks and multiple change drivers. The use of practices reduces combinatorial effects between subsystems but does not eliminate these and therefore the result is not a NS. Mannaert et al. (2011) note that more fine-grained tasks means bigger separation of tasks from each other. In this paper, we applied NS theory to coarse-grained modules (subsystems) and hence it is probably natural that we do not have a total separation of concerns in case of these. The use of the practices lays the groundwork for the design and implementation of a system as a NS but much depends on the selection of platforms and the design of more fine-grained elements of the system.

We observed that one could describe the elements of business architectures in terms of the elements of NS. Using this kind of abstraction to describe the theory was a good exercise that helped us to think about the theory. We also noticed that the description of NS theory does not distinguish understandably enough core concepts *type*, *value*, and *variable*. It is also vague in terms of what is a *change driver*. As the result it is difficult to comprehend the details of the theory. We should also not forget that maintainability that one can improve by following NS theory is only one of the quality characteristics of systems and that creating a NS does not automatically guarantee high quality of the system in terms of other characteristics. Hence, using NS theory cannot be the only mean for evaluating the quality of the system.

Future work could include elaboration of the presented metamodels. For validating NS theory as well as data-centric practices one could perform their ontological analysis in order to find possible ontological discrepancies: construct overload, construct redundancy, construct excess, and construct deficit.

Another line of research would be to investigate how application of NS theory influences different quality characteristics of different quality models (described by the ISO/IEC 25010:2011 standard). For instance, it would be interesting to know how it influences performance efficiency of the system as well as usability of the system and its specifications.

It would be interesting to more closely compare NS theory and approaches for reducing data redundancy and investigate whether the knowledge from the database field could be applied in the context of NS theory.

As was mentioned before, description of different types of systems in terms of general elements makes it possible to search deep similarities between the systems.

Yet another line of research would be investigation how much the use of the anchor modeling approach (Rönnbäck et al., 2010) would increase the normalization level of systems and whether the use of the approach is feasible in case of data-centric transactional information systems.

Finally, it would be interesting to find out to what extent the existing systems follow the principles of NS theory.

References

Bergland, G. D. (1981). A Guided Tour of Program Design Methodologies. *Computer* 14, 13–37.

Berrisford, G., Lankhorst, M. (2009). Using ArchiMate with an Architecture Method. A conversation. *Via Nova Architectura* 6.

Burns, L. (2011). *Building the Agile Database. How to Build Successful Application Using Agile Without Sacrificing Data Management*. Technics Publication, New Jersey.

Burton-Jones, A., Meso, P. (2002). How Good Are These UML diagrams? An Empirical Test of the Wand and Weber Good Decomposition Model. In: *Proceedings of ICIS*, International Conference on Information Systems (15-18 Dec. 2002, Barcelona, Spain). Paper 10.

Crain, A. (2002). Dear Dr. Use Case: Is the Clock an Actor? Rational Edge June 2002, available at http://www.ibm.com/developerworks/rational/library/content/RationalEdge/jun02/DrUseCaseJun02.pdf.

Date, C.J. (2006). *The Relational Database Dictionary. A comprehensive glossary of relational terms and concepts, with illustrated examples*. O'Reilly.

Date, C.J. (2009). *SQL and Relational Theory. How to Write Accurate SQL Code.* O'Reilly.

De Bruyn, P., Mannaert, H. (2012). Towards Applying Normalized Systems Concepts to Modularity and the Systems Engineering Process. In: Kaindl, H., Koszalka, L., Mannaert, H., Jäntti, M., Dini, P., Skala, V. (Eds.), *Proceedings of ICONS*, Seventh International Conference on Systems (29 Feb. – 5 Mar. 2012, Saint Gilles, Reunion), IARIA, 59–66.

Dumas, M. (2011). On the Convergence of Data and Process Engineering. In: Eder, J., Bieliková, M., Tjoa, A.M. (Eds.), *Proceedings of ADBIS*, 15th East-European Conference on Advances in Databases and Information Systems (20-23 Sept. 2011, Vienna, Austria), LNCS 6909, Springer, Berlin Heidelberg, 19–26.

Eessaar, E. (2014a). Specifying Business Architecture as a Step Towards Achieving Normalized Systems. In: Haav, H.-M., Kalja, A., Robal, T. (Eds.), *Proceedings of Baltic DB&IS*, 11th International Baltic Conference on DB and IS (8-11 June, 2014, Tallinn, Estonia), Tallinn University of Technology Press, 425–432.

Eessaar, E. (2014b). A Set of Practices for the Development of Data-Centric Information Systems. In: Escalona, M.J. et al. (Eds.), *Proceedings of ISD*, 22nd International Conference on Information Systems Development (2-4 Sept. 2013, Sevilla, Spain), Springer (in press), DOI: 10.1007/978-3-319-07215-9_6.

Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Menlo Park, Calif.

Green, D.G., Leishman,T. (2011). Computing and complexity – networks, nature and virtual worlds. In Hooker, C.A., Gabbay, D.M, Thagard, P., Woods, J. (Eds.): *Philosophy of Complex Systems*, Vol. 10 (Handbook of the Philosophy of Science), North Holland, 137–162.

ISO/IEC 25010:2011 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models, first ed.: 2011-03-01.

Larman, C. (2002). *Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design and the Unified Process*. 2nd ed. Prentice Hall, Upper Saddle River, NJ.

Mannaert, H., Verelst, J., Ven, K. (2011). Towards evolvable software architectures based on systems theoretic stability. *Softw. – Pract. and Exp.* 42, 89–116.

Ould, A.M. (1997). Designing a re-engineering proof process architecture. *Business Proc. Man. J.* 3, 232–247.

Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems Into Modules. *Communic. of the ACM* 15, 1053–1058.

Piho, G. (2011). *Archetypes Based Techniques for Development of Domains, Requirements and Software. Towards LIMS Software Factory*. PhD thesis, Tallinn University of Technology, Tallinn, Estonia.

Roost, M., Kuusik, R., Veskioja, T. (2001). A Role-Based Framework for Information System Self-Development. In: Russo, N.L., Fitzgerald, B., DeGross, J.I. (Eds.), Proceedings of IFIP

TC8/WG8.2 Working Conference (27-29 July 2001, Boise, Idaho, USA), Kluwer Academic Publisher, Norwell, MA, USA, 95–105.

Roost, M., Kuusik, R., Rava, K., Veskioja, T. (2004). Enterprise Information System Strategic Analysis and Development: Forming Information System Development Space For Enterprise. In: Okatan, A. (Ed.), *Proceedings of ICCI*, International Conference on Computational Intelligence (17-19 Dec. 2004, Istanbul, Turkey), Turkey, 215–219.

Rönnbäck, L., Regardt, O., Bergholtz, M., Johannesson, P., Wohed, P. (2010). Anchor Modeling —Agile Information Modeling in Evolving Data Environments. *Data & Knowl. Eng.* 69, 1229–1253.

Sanchez, R., Mahoney, J.T. (1996). Modularity, flexibility, and knowledge management in product and organization design. *Strat. Man. Journal* 17, 63–76.

Sanz, J.L.C. (2011). Entity-Centric Operations Modeling for Business Process Management – A Multidisciplinary Review of the State-of-the-Art. In: Gao, J., Lu, X., Younas, M., Zhu, H. (Eds.), *Proceedings of SOSE*, 6th IEEE International Symposium on Service Oriented System Engineering (12-14 Dec. 2011, Irvine, CA, USA), IEEE, Piscataway, NJ, 152–163.

Verelst, J., Silva, A.R., Mannaert, H., Ferreira, D.A., Huysmans, P. (2013). Identifying Combinatorial Effects in Requirements Engineering. In: Proper, H.A., Aveiro, D., Gaaloul, K. (Eds.), *Proceedings of EEWC*, Third Enterprise Engineering Working Conference (13-14 May 2013, Luxembourg), LNBIP 146, Springer, Berlin Heidelberg, 88–102.

Wand, Y., Weber, R. (1990). An Ontological Model of an Information System. *IEEE Trans. on Software Eng*. 16, 1282–1292.

Wand, Y., Weber, R. (1995). On the deep structure of information systems. *Inf. Syst. Journal* 5, 203–223.

Weber, R. (1997). *Ontological Foundations of Information Systems*. Coopers & Lybrand and Accounting Association of Australia and New Zealand, Melbourne.

Authors' Information

E. Eessaar, dr., is a full-time Associate Professor at the Department of Informatics in Tallinn University of Technology. He teaches courses about database design and database development. He is the author or a co-author of about 40 research papers and the author of one book in the field of databases and information systems development. Research interests: data models, model- and pattern-driven development and evolution of information systems (including databases) with the help of domain-specific languages, metamodeling, metadata, and software measures.

Received September 9, 2014, accepted September 10, 2014.