Bus Inf Syst Eng 60(1):55–67 (2018)
https://doi.org/10.1007/s12599-017-0510-4

RESEARCH PAPER

Enabling Normalized Systems in Practice – Exploring a Modeling Approach

Peter De Bruyn · Herwig Mannaert ·
Jan Verelst · Philip Huysmans

Received: 28 February 2017 / Accepted: 4 October 2017 / Published online: 11 December 2017

Abstract Contemporary organizations are required to adapt to a changing environment in an agile way, which is often deemed very challenging. Normalized Systems (NS) theory attempts to build highly evolvable software systems by using systems theory as its theoretical underpinning. A modeling method which supports the identification of the NS elements, required for building NS sofware in practice, is currently missing. Therefore, the paper introduces an approach for creating both data models and processing models in the context of NS, as well as their integration. It is discussed how these models can be taken as the input for the actual creation of evolutionary prototypes by using an earlier developed supporting tool. The modeling approach and its suitability for feeding the tool are evaluated to discover their current strengths and weaknesses.

Keywords Normalized systems · Data modeling · Process modeling · Agility

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s12599-017-0510-4) contains supplementary material, which is available to authorized users.

Accepted after two revisions by the editors of the special issue.

P. De Bruyn (✉) · H. Mannaert · J. Verelst · P. Huysmans
Department of Management Information Systems, Faculty of Applied Economics, University of Antwerp, Prinsstraat 13, 2000 Antwerp, Belgium
e-mail: peter.debruyn@uantwerpen.be

H. Mannaert
e-mail: herwig.mannaert@uantwerpen.be

J. Verelst
e-mail: jan.verelst@uantwerpen.be

P. Huysmans
e-mail: philip.huysmans@uantwerpen.be

1 Introduction

As their competitive environment changes rapidly, organizations have to adapt themselves accordingly in an agile way. This puts evolvability requirements on every company in all its facets including its strategy, enterprise models, software systems, etc. Obtaining agility at the IT level is challenging and consistently ranked as one of the top 3 organizational IT management concerns (Kappelman et al. 2014). However, the amount of information systems (IS) research directed towards the creation of evolvable systems is marginal (Agarwal and Tiwana 2015): most research is exclusively directed towards the initial phases of the IS development life cycle whereas the majority of the costs for building and maintaining systems is situated in the later phases.

Normalized Systems (NS) theory attempts to create highly evolvable software systems by proposing a set of theorems which are formally proven to be necessary conditions to obtain evolvability (Mannaert et al. 2011, 2016). The theory exhibits several appealing and unique characteristics for tackling the agility challenge: it is grounded in concepts from systems theory, the derived theorems provide specific programming guidance while unifying multiple software design best practices, and it has proven its feasibility in practice by multiple implementations for different types of systems in industry (Mannaert et al. 2012; van der Linden et al. 2017; Eessaar 2016).

While NS software is able to provide evolvability at the software level, the question on how to model an enterprise so that it can be supported and operationalized by NS software has currently remained unsolved. A first goal of this paper is therefore aimed towards the creation of such a modeling method. This endeavor is not trivial. To begin with, as the NS theorems imply a very dense and highly structured code base (which is hard and time-intensive to obtain by manual coding), it is advised to generate a large portion of the code by the recurrent use of a set of NS "elements" (Mannaert et al. 2012, 2016). As a consequence, a good modeling method should allow the adequate representation of enterprise requirements in terms of these NS elements. Additionally, enterprise requirements which are themselves modeled in such a way that they are contradicting with NS theorems could still nullify the overall evolvability of an enterprise. A good modeling method should consequently also consider possible implications of NS at this level. A second goal of this paper is to illustrate how the output of this modeling effort can be used as an input to feed a supporting tool which was earlier created to drive the generation of the structured code base, based on the NS elements. The feasibility of a modeling approach in terms of NS elements and the synergies with a supporting tool are shown. We illustrate how this can simplify the initial phases of the development process of NS software and further improve the practical impact of the theory.

The remainder of this paper is structured as follows. In Sect. 2, we provide background information regarding NS. Sections 3 and 4 describe, respectively, a modeling approach and the resulting use of the supporting tool. Section 5 discusses an evaluation of our approach based on our current experiences. Our final conclusions are offered in Sect. 6.

2 Normalized Systems Theory

Changing and adapting their supporting software systems is crucial for organizations in order to remain agile and flexible. However, several indications exist that this is hard to realize in practice. For instance, Lehman's Law of Increasing Complexity states that "as an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it" (Lehman 1980, p 1068). This deteriorating structure implies that applying similar changes to a system becomes more difficult over time. NS tries to tackle this phenomenon by demanding BIBO (bounded input, bounded output) stability as defined in systems theory to software systems (Mannaert et al. 2011, 2016). This means that the impact of a bounded set of changes to a software system should only depend on the type of the changes, not on the size of the system to which they are applied. Conversely, changes which are dependent on the size of the system are coined *combinatorial effects* (Mannaert et al. 2011, 2016). In an agile environment, where we can assume that software systems are ever growing and changing, such combinatorial effects become eventually prohibitive as their effort (and therefore, cost) may become larger than the cost for creating an entirely new software system. A system which is free of combinatorial effects is called a *Normalized System* (Mannaert et al. 2011, 2016).

2.1 NS Theorems and Elements

NS proposes four *theorems*, which have earlier been proven to be necessary conditions in order to avoid combinatorial effects (Mannaert et al. 2011, 2016):

– *Separation of Concerns*, stating that a processing function can only contain a single concern (i.e., change driver or each part which can independently change);
– *Data version Transparency*, stating that a data structure that is passed through the interface of a processing function needs to exhibit version transparency (i.e., not impacting processing functions in case it is updated);
– *Action version Transparency*, stating that a processing function that is called by another processing function needs to exhibit version transparency (i.e., not impacting its calling processing functions in case it is updated);
– *Separation of States*, stating that the calling of a processing function within another processing function should exhibit state keeping (i.e., before calling another processing function, a state should be kept).

Applying these theorems leads to very fine-grained modular systems, which are very hard to create by human programming. Therefore, 5 higher-level detailed design patterns (so-called *elements*) have been proposed to create NS software in practice (Mannaert et al. 2012, 2016), each aggregating and encapsulating a set of software constructs and providing the basic functionality of an information system:

– *data elements*, to represent data variables and structures, and including support for cross-cutting concerns such as remote access and persistence support;
– *task elements*, to represent processing instructions, and including support for cross-cutting concerns such as remote access, logging and access control;
– *flow elements*, to handle control flow and orchestrations (i.e., the execution of a number of task elements on a specific target data element in a stateful way);
– *connector elements*, to allow the interaction with external systems (via a user interface or another application);
– *trigger elements*, to offer periodic clock-like control and checking whether a task element needs to be triggered.

Each of these elements provides, for a particular functional requirement (e.g., representing data, executing a calculation), a transformation into software code. That means that a set of cross-cutting concerns (e.g., data access control, persistency, etc.) is embedded and generated with each element instantiation. The elements have been proven to be free of combinatorial effects against a predefined set of anticipated changes (Mannaert et al. 2016). Therefore, as these elements allow to provide all basic functionalities of contemporary software systems, NS software can be built by systematically instantiating the above mentioned elements as required. Moreover, it has been argued that NS reasoning is applicable to modular structures in general (De Bruyn 2014; Mannaert et al. 2016).

2.2 The NS Development Trajectory

Figure 1 provides a schematic overview of the system development trajectory for NS software. First, real-world functional requirements have to be transformed into instantiations of the 5 NS elements, indicated by arrow 1. While the different requirements from the stakeholders can (optionally) initially be captured by means of high-level analysis techniques (e.g., use cases), the eventual analysis should result into a more low-level system requirement specification in terms of the NS elements (i.e., the domain model). Next, as indicated by arrow 2, each of these NS elements is transformed into software code by using the design patterns which were specified for each element. This part of the system development lifecycle is highly structured and results in a large amount of code, which can be used as a fully operating prototype with standard out-of-the-box functionalities. Afterwards, plug-in code can be applied to the generated code to provide specific functionalities which are not embedded by default (e.g., a particular validation, business rule, calculation, etc.) and to convert the prototype into a production ready application, which is indicated by arrow 3. The application based on the domain model, potentially enriched with plug-in code, can be shown to actual end users (indicated by arrow 4). The whole procedure is typically highly iterative as it allows to incrementally enlarge the scope covered by the application, show alternative implementations to clarify conflicting visions (e.g., different stakeholders from different organizational units), and verify whether the high-level end user needs were correctly translated into an NS application. The modeling approach proposed in Sect. 3 mainly supports the identification of the NS elements or domain model (cf. arrow 1). Specifying this domain model within the supporting tool (as described in Sect. 4) supports the transformation of the domain model into software code (cf. arrow 2). The earlier developed tool equally allows the overall management of plug-in code (cf. arrow 3) and the validation of user expectations (cf. arrow 4).

3 Modeling Approach

As stated before, an NS modeling approach should ultimately deliver an identified set of NS elements, to be transformed into code afterwards. We are aware that functional requirements in this form are formulated in a very detailed and low-level way but stress that any other analysis approach aiming to deliver actual working software should, at some point in time, provide a similar level of detail. We explicitly aim to leverage existing IS modeling techniques and enrich or restrict them with the purpose of identifying NS elements. Based on NS and its generalization to modular structures (Mannaert et al. 2016; De Bruyn 2014), several design implications can be derived for the modeling of functional requirements, which we will use to present an NS compliant modeling approach.

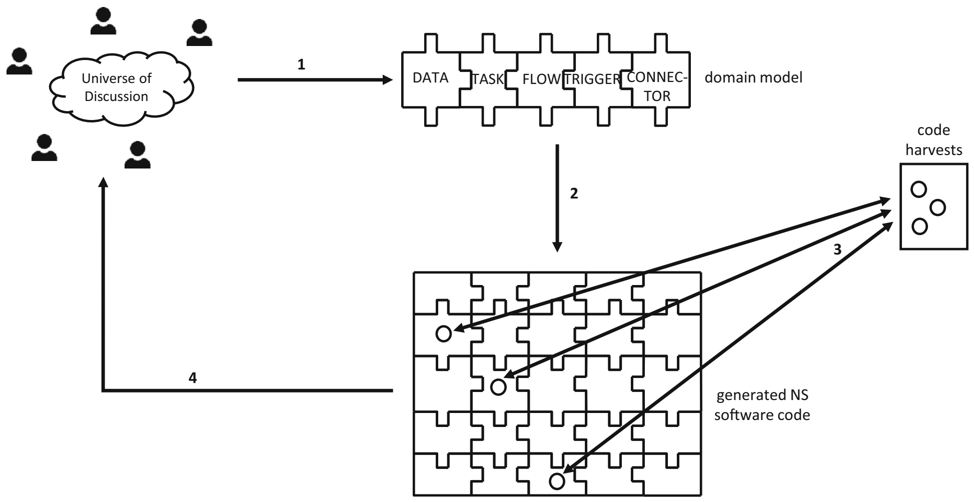

Fig. 1 The NS system development trajectory

As a consequence, our approach embeds certain design principles into the enterprise modeling procedure.

Consider for instance the Separation of Concerns theorem. While functional requirement models should provide clarity regarding data requirements (which information is to be captured) and processing requirements (e.g., the execution of algorithms), the theorem implies that data and processing requirements should be modeled in separate modeling constructs. Indeed, data entities (e.g., information regarding an invoice) and related processing functionalities (e.g., calculating the relevant amount of VAT) can independently change and therefore constitute separate change drivers or concerns. Mixing both would make a modeling construct subject to multiple change drivers and might therefore hinder the smooth incorporation of changes within a domain model (e.g., a single mandatory change is then likely to impact multiple locations in the domain model). This separation should however not suppress the need to integrate both requirement dimensions, as we will discuss later on.

As a separation of data and processing modeling constructs is required, we will now discuss each of them consecutively. Each time, we reflect on the specific modeling implications for them based on NS and provide a brief example. We also reflect upon related work and how that differs with our approach. Afterwards, we consider the integrated enterprise model. As the elements provide a standard implementation of cross-cutting concerns (cf. supra), we consider further technical specifications in this regard out of scope. As the design patterns of the data/task/flow elements automatically generate their supporting connector and trigger elements (whenever required) as well, the explicit connector and trigger element specification within the domain model is not required.

3.1 Data Model

We propose a variant on entity-relationship diagrams (ERD) for modeling data requirements in order to allow the required separation between data and processing modeling constructs. This modeling will allow us to identify the required NS data elements for an application.

3.1.1 NS Theorem Implications

First, the separation of concerns theorem implies that enterprise modeling and technical implementation details are different types of concerns and should therefore not be mixed in the same domain model either. The design patterns used to generate the code for the NS elements take care of many technical issues (e.g., access control, logging) and the domain model should therefore limit itself to business oriented and even exclusively anthropomorphic entities (i.e., modeling in terms of items having the form of or being similar to human natural language). For instance, data entities managing access control or logging cannot be embedded in an NS domain model. Second, the same theorem suggests that core modules and non-core (cross-cutting) modules should be distinguished. We therefore propose a set of data entity dimensions which acknowledge the existence of several types of modules at the level of data requirements (De Bruyn et al. 2016):

- *core entities*: the most essential “things” of an enterprise (often the most important nouns in a textual description of the universe of discussion or concepts which are crucial in its business model), which might sometimes also be subject to dynamic behavior (i.e., the need of being processed or being changed of state, in which case the entity carries a “status” field to keep track of the state);
- *taxonomy entities*: entities which are used to classify (i.e., apply a taxonomy to) core data entities (within the application domain) into categories or groups in order to grasp and lower the complexity of the world;
- *directory entities*: entities tracking the place of a core entity within a certain directory system such as building, affiliations, closets, etc.;
- *domain entities*: entities with generally accepted information about a general domain (e.g., regarding countries, industries, fiscal codes, etc.) which are typically required and reused in many applications (therefore not specifically related to the application domain);
- *integration entities*: entities which are required for the functioning of the domain model under consideration but are provided by another application (internal or external) with which, as a consequence, integration is required.

Third, the Version Transparency theorems require that the dependencies between the (modeling) constructs are made explicit. This can be operationalized by specifying the relationships between the data entities as well as their multiplicities: one-to-one, one-to-many or many-to-many. Further, maintaining version transparency implies that entities on which other entities depend cannot be removed. For instance, in a situation where a Phone entity has a many-to-one relation with a PhoneType entity (work, private), this PhoneType entity cannot be deleted as long as the Phone entity exists (as in traditional cases, the Phone entity will keep a reference to the PhoneType entity). As version transparency at the technical level is handled by the NS elements during software generation, other implications (e.g., coping with additional or removed fields without combinatorial effects) are automatically dealt with.

As the Separation of States theorem only holds within a processing perspective, no specific design implications can be derived from it within the context of a data model.

3.1.2 Illustration

Figure 2 shows a visual illustration of our proposed data modeling approach with its data entity dimensions, for the case of a fictitious and simplified car rental business. The core entities are indicated in blue (Car, Rental, Booking, Person, CarPickup, CarDropOff), taxonomy entities in red (CarModel, CarBrand, CarCategory), directory entities in green (Location, FleetLocation, ParkingLot), domain entities in purple (City, Country) and integration entities in orange (Email, Phone, SocialMediaAccount). Each of these data entities will correspond to an NS data element when building an application for the rental company.

3.1.3 Contrast with Related Work

In terms of modeling notations, ERD data diagrams have a long tradition in data modeling (Chen 1976). Clearly, our model from Fig. 2 could well be presented in other notations as well, such as a ORM model (Halpin 2001) or UML domain class diagram (Booch et al. 1999). The latter would then need to contain only member variables given our separation between data and processing. In terms of prescriptive guidelines, the seminal work of Codd (1970) describes how to design a data model without redundant data. Some general data model patterns or reference models (see e.g., Hay 1996; Silverston 2001; Scheer 1998) provide best practice data model templates for several domains, although their adoption rate is unclear. Exclusively considering business relevant (i.e., non-technical) entities within a data model has been suggested previously by several authors, including contributions within domain-driven design (Hruby 2006; Evans 2003). We are not aware of earlier work on theoretically based prescriptive guidelines specifically aimed towards the optimization of the evolvability or adaptability of data models, although Moody (2003) listed easy incorporation of changes within data models (labeled as flexibility) as a desired quality characteristic of data models.

We therefore believe that our proposed approach regarding data models differs from existing work in at least the following ways. First, a categorization as the one we propose is, to the best of our knowledge, new. The categorization might assist analysts in more exhaustively identifying required entities and suggests an additional level of structure (i.e., core data entities with their surrounding non-core data entities). Second, our data modeling approach is tuned towards the creation of evolvable systems and the implementation by NS in particular. Third, we advocate to the use of existing data modeling notations but restrict them to only one basic data entity construct and its relationships: no advanced constructs such as

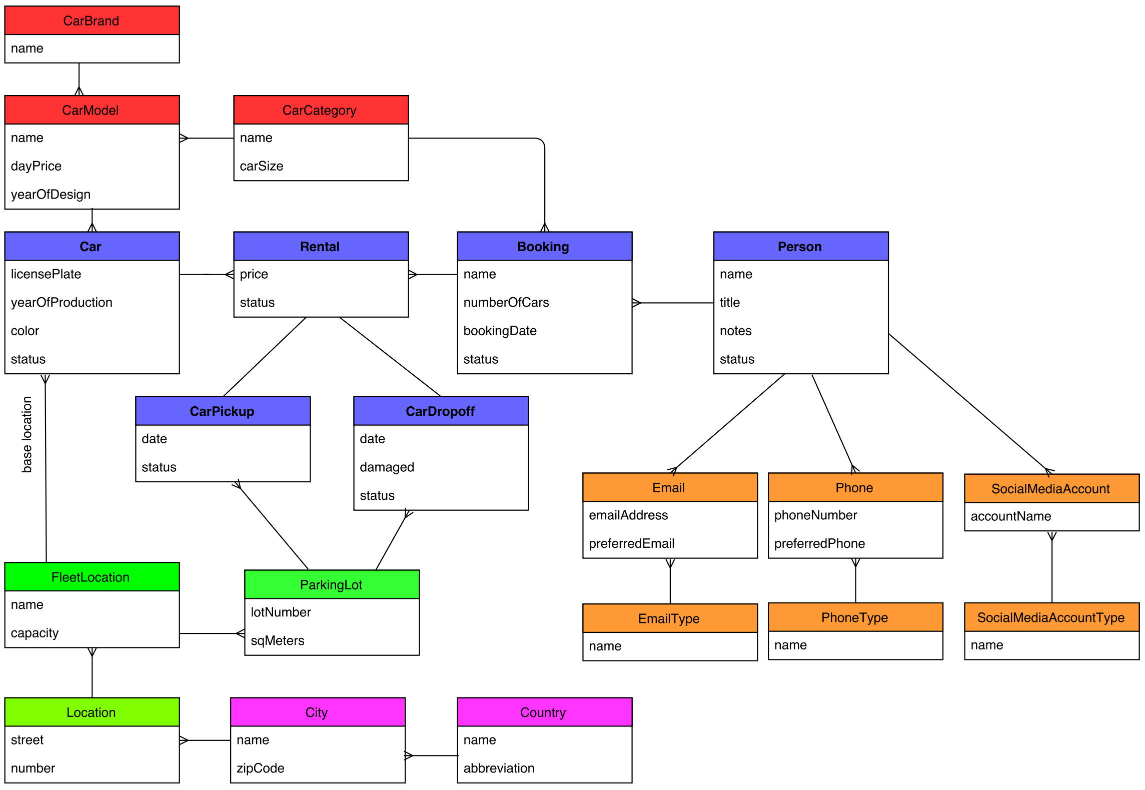

Fig. 2 Data Model

inheritance (as often employed in object-oriented analysis) are allowed as they can be detrimental for the agility of the data model. Suppose for instance that a hierarchical refinement of the data entity Person is made based on nationality type: a set of common attributes would be specified in the general Person entity (e.g., first name and last name) which would be inherited by its child entities (BelgianPerson, AmericanPerson, etc.) that hold, at their turn, also attributes for their specific specialization (e.g., a SIS number for Belgian Persons). Suppose further that later on, a distinction has to be made based on age type (junior/senior), or gender type (male/female), or education type (none/high school/university). The specific attributes for these specializations should be duplicated in every already existing branch (here: each nationality), which is a combinatorial effect as the impact is depending on the size of the system and is resulting in highly non-anthropomorphic entities such as BelgianJuniorPerson, AmericanSeniorPerson, etc. In addition, when aiming for evolvability, it is advised to avoid inheritance from a technical software point of view as well (Mannaert et al. 2016). Instead, we advice to only employ standard data entities for each type within each refinement dimension (BelgianInformation, AmericanInformation, MaleInformation, etc.). Via exclusive OR restrictions, an entity can then be refined according to several dimensions without introducing combinatorial effects.

3.2 Processing Model

We propose a variant on state machines for modeling processing requirements, which aligns with the required separation between data and processing modeling constructs. This modeling will allow us to identify the required NS task and flow elements for an application.

3.2.1 NS Theorem Implications

First, the Separation of Concerns theorem implies that the tasks of a state machine (e.g., a specific calculation) and the flow of a state machine (defining the sequence, iteration, selection of tasks) are different concerns and should be modeled in different constructs. Indeed, each task (an individual calculation) as well as the flows (the order in which tasks are to be performed) can be changed independently and therefore constitute separate change drivers or concerns. As we defined core data entities as entities which are susceptible for dynamic behavior (in which case they carry a “status” field), tasks and flows operate on core data entities. Their status field allows the appropriate task in the flow to be initiated and will be updated after the completion of each task. The Separation of Concerns theorem further implies that every task or flow should only operate on one single argument data entity as otherwise, again, multiple change drivers would unnecessarily be combined. The same theorem demands that every processing unit which can independently change is to be identified as a concern and should thus be modeled as a separate task. Therefore, in general, this gives rise to a first set of task types:

– *Standard task*: the information system itself performs an actual action, e.g., sending an email, checking the availability of a part, calculating the VAT, etc.;
– *Manual task*: a human user is required to perform the action, and to indicate its completion through a user interface (e.g., approving an expense report). Every task executed by another human actor is considered as a separate task. In case of collaborative tasks, composite human actors are considered;
– *External task*: the action is assumed to be performed outside the considered flow, possibly even within another information system (e.g., triggering an alarm).

Second, the Separation of States theorem implies that flows should be performed in a stateful way. Therefore, we propose to model tasks and states (and therefore, the relevant state transitions) separately.

Third, the Version Transparency theorems require that the dependencies between the (modeling) constructs are made explicit. As the scope of the genuine processing embedded within one flow should be limited to only one core data entity, many fine-grained flows are identified which need to be able to interact. This gives rise to a second set of task types:

– *Bridge task*: the information system creates one or multiple instances of another (core) data element that will be processed in its own flow (in case it has one operating on it), e.g, creating an order upon an approved offer;
– *Update task*: in case a bridge task creates one or multiple instances of another data element, it might be required to keep the parent element automatically up-to-date regarding the child elements (e.g., all parts of the order are available now). An update task enables such automatic “updating”.

Further, maintaining version transparency implies that data entities or flows on which other flows depend cannot be removed. For instance, when an order flow needs to get updates from all its associated parts, this part element needs to remain existing. As version transparency at the technical level is handled by the NS elements during software generation, other implications (e.g., implementing a new task version without creating combinatorial effects) are dealt with at this level.

3.2.2 *Illustration*

Figure 3 shows a visual illustration of our proposed process modeling approach for the case of the processing requirements regarding a Booking within a fictitious and simplified car rental business. That is, it concerns a flow operating on one core data element (in this case: Booking). This booking flow consists of multiple tasks, having different types being indicated by means of the letter at the right upper corner of each task box: *S* for standard task (PersonRetriever, PaymentChecker, CarReserver), *M* for manual task (BookingConfirmer), *B* for bridge task (PersonCreator, RentalsCreator) and *U* for update task (RentalChecker). As we consider a small and simplified case, no external task (*E*) is present in this example. Remark from Fig. 2 that the Booking core data element has a status field in which the current state of each Booking instance (and the flow operating on it) is kept (going from Created to PersonExisting to AdvancePaid, etc.). Similarly, flows can be modeled for the processing activity regarding Persons, Rentals, etc. Each flow will correspond to an NS flow element and each of their tasks will correspond to an NS task element when building an application for the rental company.

3.2.3 *Contrast with Related Work*

In terms of modeling notations, our processing models could clearly be represented in UML state diagrams (Booch et al. 1999) or Event-driven Process Chains (Keller et al. 1992) as they allow the indication of tasks and states. BPMN diagrams would be possible as well, although it is advised to include states in the notation one way or the other. In terms of prescriptive guidelines, a set of seven process modeling guidelines (7PMG) has been proposed by Mendling et al. (2010) and Moreno-Montes de Oca and Snoeck (2014) distilled a set of 27 guidelines based on a broad set of sources. Most of these guidelines are directed towards the improvement of the general understandability of business processes and are of a pragmatic nature. Others focus on the engagement of stakeholders by proposing certain project management guidelines (e.g., Sammon et al. 2016) or propose comparison and integration methods for sets of processes (e.g., Xiao and Zheng 2012; de Cesare and Serrano 2006). Some authors have argued that the qualitative nature of business processes tends to discourage formal or algorithmic approaches (Vergidis et al. 2007) and that specific design rules on how to modularize (i.e., design) business processes are largely missing (Reijers and Mendling 2008) which can be considered problematic (Becker et al. 2000). Therefore, guidelines specifically directed towards a specific optimization criterion such as the evolvability of business processes are scarce (Van Nuffel 2011). For instance, searches on “evolvable ‘business processes’”, “evolvability ‘business processes’” and “flexibility ‘business processes’” in the Web of Science in June 2017 returned 0, 0 and 26 hits, respectively (the search on flexibility often containing work on how to deal with deviating process instances).

We therefore state that our approach differs from existing work in that we provide a theoretically based modeling and design approach to enable evolvability in the context of processing functionality (particularly in an NS context). Similar to Sect. 3.1, we advocate the use of existing processing modeling notations but restrict them to only one basic data flow construct (with a particular sequence of tasks), operating on one argument data entity: no heterogeneous business processes spanning multiple data entities are allowed. Such design is detrimental for the agility of the processing model as each business process is in that case dependent on multiple change drivers which may result in combinatorial effects.

3.3 An Integrated Enterprise Model

3.3.1 *Overview*

While we started this section by stating that a clear separation is required between data and processing modeling constructs, this does not mean that they are not related to one another or should not interact. Figure 4 visualizes an integrated enterprise model resulting from our NS based approach. Here, three planes can be discerned:

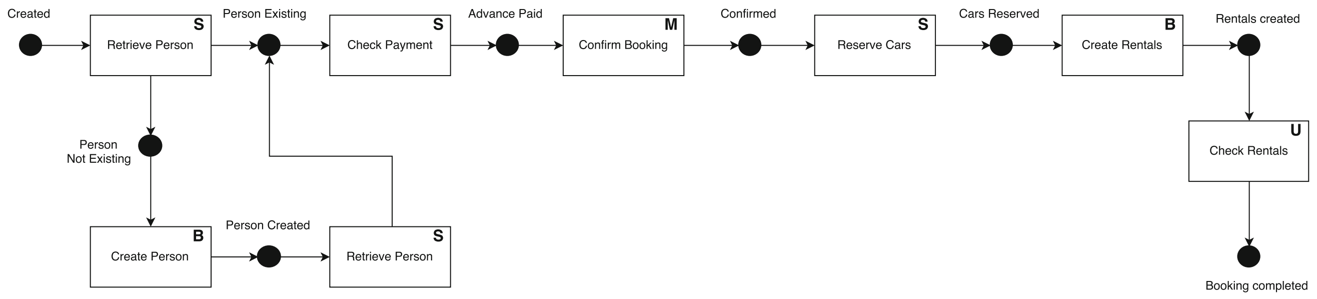

Fig. 3 Processing model

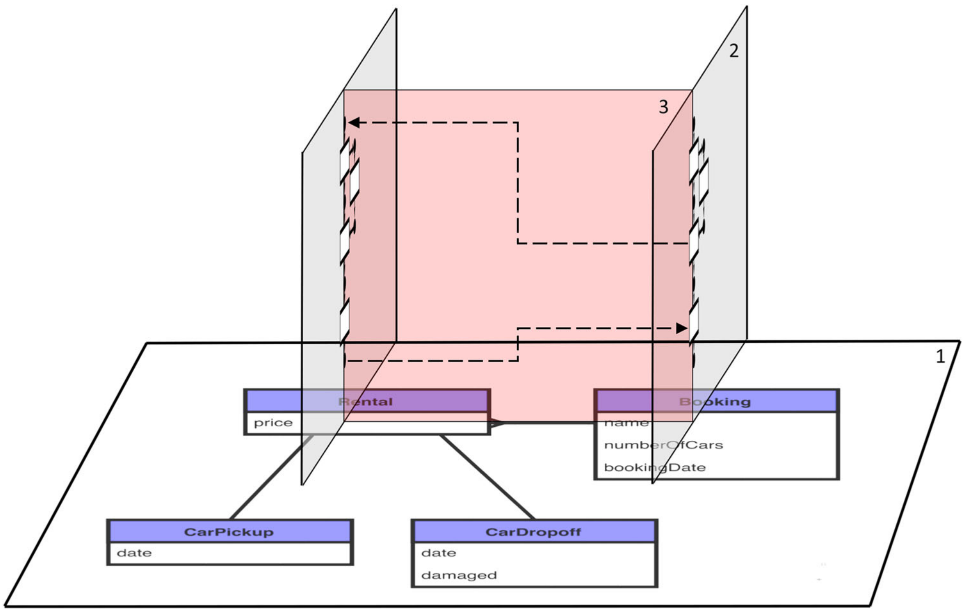

Fig. 4 Integrated enterprise model

– the *horizontal plane* (indicated by number 1), depicting (a subset of) the data entities;
– the *vertical plane* (indicated by number 2), depicting (a subset of) the flows operating on core data entities;
– the *plane perpendicular* (indicated by number 3) to both the horizontal and vertical plane, depicting the dependencies between the data entities and flows.

The figure assists in explicitly clarifying several characteristics of our approach. First, the figure reflects the application of the NS theorems (enabling evolvability) during our modeling efforts. The horizontal and vertical plane provides evidence of the applied Separation of Concerns regarding data and processing. Within the vertical plane, the stateful sequencing of tasks within each flow provides evidence of the applied Separation of States theorem. The perpendicular plane expresses the dependencies which should be taken into account in order to guarantee Version Transparency. Second, the figure highlights the strict and systematic integration of our models. The horizontal and vertical plane show that each flow (consisting of one or multiple tasks) operates on one and only one core data entity. The perpendicular plane shows that some data entities are dependent on other data entities by multiplicity relationships (see Fig. 2), while some flows are dependent on other flows by bridge or update tasks. It is interesting to remark that this plane relates to both the data entities and flows in the same way. Stated otherwise, the dependencies between data and processing occur in a parallel way: indeed, bridge and update tasks represent tasks working on a core data entity and typically connecting to another data entity with which that entity has a one-to-many (or many-to-one) relationship. Therefore, our approach avoids – by definition – a schism between data and process modeling. Finally, the figure illustrates the possibility for a gradual way of working. Without the need to tackle the full complexity of an enterprise (universe of discussion) at once, an analyst can first create (a subset of) the data model as shown in the horizontal plane: typically, some core data entities are identified at the start which are later on supplemented with non-core data entities. Next, some flows operating on them can be added (the vertical plane). At all times, the dependencies should be managed appropriately (the perpendicular plane). Later on, this model can be gradually extended with some additional data entities, flows, tasks, etc. Such way of working is expected to be more difficult in case no categorization of data entities is performed or processing requirements are purely modeled on the basis of their current chronological order in isolation from the data model. Therefore, our approach is believed to result in an enterprise (domain) model which can more easily cope with change.

3.3.2 *Contrast with Related Work*

Studying the integration between data and processing views on information systems is obviously not new. For instance, already Jackson Structured Programming (JSP) was a ‘data first’ approach in which the structure of the program was advised to be aligned with the structure of data processing (Jackson 1975). In the object-oriented paradigm, data and action are even “integrated” at the construct level of a class. Interaction between constructs is then visualized via, for instance, UML interaction diagrams (Booch et al. 1999). The concept of coupling has been put forward as a way to operationalize dependencies between constructs (Yourdon and Constantine 1979) and has been applied to evaluate several design alternatives (Larman 1997). And related to our flow conceptualization as working on data entities, it needs to be mentioned that several entity-centric modeling approaches exist (see e.g., Sanz 2011) and parallels with early entity life histories can be drawn (Jackson 1975). Therefore, we believe that our integration approach is not necessarily new in terms of modeling notation or general rationale, when compared to existing work. Rather, the unification of the 3 different dimensions (planes) as discussed above in alignment with NS theorems is deemed to provide a new perspective.

4 Tool Support

A supporting tool was developed earlier to drive the generation of NS software. We discuss how the results of our proposed modeling approach can be used by analysts as input for that tool, i.e., how our models in terms of NS elements can be transformed into working NS software (prototypes). Later on, these prototypes can be further developed into production ready applications incorporating organization specific plug-in code.

4.1 Prototype Definition

In order to enter the enterprise model in the tool, an analyst should define a new application. Next, several components can be added to this application: typically, several standard components (providing default functionality such as user management) and one application specific component are added to the application. Within this application specific component, the domain model can be specified. That is, the different data, task and flow elements can be created as shown in Fig. A1 in the appendix (available online via http://springerlink.com). The data elements correspond to the entities in the data model (e.g., Fig. 2), the task elements to the square boxes in the processing model (e.g., the squares within Fig. 3) and the flow elements to the flows in the processing model (e.g., the state transitions as visualized within Fig. 3). For each data element, fields (i.e., attributes within the data model) can be specified as well as their value types (text, number, link to other fields), options (e.g., display options) and finders (search operations). For each task element created, the target data element can be indicated (i.e., a blue core data entity) as well as its target class (empty test code or an actual task implementation) and options (e.g., whether it is a bridge task or not). For each flow element created, a target data element should be specified, together with its corresponding status field (reflecting the changing state of an instance of a data element) and a set of state transitions (defining which state – the black dots from Fig. 3 – results in which other state – another black bullet – by the execution of which task element, being defined above). As connector and trigger elements are automatically generated when needed for data/task/flow elements, their manual specification is not required. Remember from Sect. 2 that each element will be provided with a set of cross-cutting concerns (such as security, remote access, etc.) during code expansion (providing typical non-functional requirements). This means that all mandatory information to generate a prototype can be directly derived from the data and processing model presented in Sect. 3: it is merely a formal equivalent of the graphical representations from Figs. 2, 3 and 4. Remark that this model specification is technology agnostic and thus fully remains within the role of the analyst as domain expert.

For instance, when considering an application for a typical car rental business, the data elements as identified during data modeling efforts are entered in the tool as can be seen in Fig. A1: Booking, Car, CarDropoff, Person, Rental, etc. Further, one can notice that for each data element, such as the Booking element in Fig. A1, the relevant fields (bookingDate, numberOfCars, etc.) are defined. Within the tabs "TaskElement" and "FlowElement" the identified task elements (RetrievePerson, CheckPayment, ConfirmBooking) and flow elements (BookingFlow) are entered similarly.

4.2 Prototype Generation

After the domain model has been entered, an application instance can be created. Here, the GUI framework (e.g., a combination of bootstrap and knockout) can be chosen, as well as the type of application server (e.g., Java EE application server) and the underlying database (e.g., HSQL). Based on these application instance details, the standard code base for the application can be generated (i.e., "expanded") and inspected by the tool (see Fig. A2 in the appendix). For instance, when considering the car rental business application, more than 30 java classes (e.g., BookingAgent, BookingBean, BookingClient, BookingCruds) will be generated for each defined data element in the application, together with the required database schemas as well as generated html and jsp files for inspecting and modifying information regarding Bookings, Rentals, etc. (a more detailed overview of this code generation process can be found in Mannaert et al. 2016).

The code can subsequently be compiled (i.e., "built") and deployed. This results in a usable prototype of the specified domain model. Some of the current out-the-box features include CRUD functionality for every data element via a standard web page, clickable linkages between data elements, search functionalities on upfront specified fields, import clients, report generation and a master-detail overview for all one-to-many data elements throughout multiple levels. Flows defined on core data elements can be illustrated by creating an instance of that element with a status field value corresponding to the start status of the flow. By doing so, this status will be detected by the engines, which will make the data element go through the different states in its lifecycle (only performing genuine business functionalities when manually added). For instance, when considering the car rental business application, the prototype will allow users to perform CRUD operations on Bookings, Rentals, etc. via a GUI in the browser. Search operations can be executed (e.g., which Rentals had a price higher than 250 euros?), reports can be printed (e.g., all detail information regarding a person as well as his related bookings and rentals) and master-detail screens (e.g., showing the information of all Rentals associated with a particular Car in one screen) can be inspected.

This prototype can be shown to potential end users and test data can be entered. Based on this operationalization of the enterprise model, end users can explore and experiment with the application. Missing or misunderstood requirements can be communicated to the analyst who can iteratively refine the domain model and re-expand the application until it complies with the actual enterprise model.

4.3 Addition of Plug-in Code

After initial prototypes have been created and refined, the domain model (and therefore, initial prototype) can be shared with the developers. They can add plug-in code between predefined "anchors" within the generated code, for instance to implement the genuine processing functionality of a certain task or to customize the lay-out of a generated CRUD screen. This code can be compiled together with the automatically generated code but can also be set aside ("harvested") and re-inserted afterwards ("injected") in case a new version of the domain model is expanded so that these customizations do not get lost. For instance, when considering the car rental business application, validations should be performed (e.g., the bookingDate should be in the future) and the tasks (e.g., CheckPayment) should be implemented. As the tool provides a structured overview of all plug-in code, the analyst can monitor to which data elements or task elements customizations were added, to which features they correspond and how complex (e.g., amount of code) they were. Therefore, performing enterprise modeling conform Sect. 3 and using that as input for the supporting tool allows the analyst to manage sets of applications and their domain models, generate initial prototypes, follow-up customizations, generate advanced prototypes (up to production ready applications) and validate requirements with end users in a highly iterative way.

4.4 Contrast with Related Work

Many modeling tools regarding data and processing (Eichelberger et al. 2009; Evéquoz and Sterren 2011) exist, some of them allowing our advocated separation of data and processing functionality. Also, several tools have been created before which generate code based on specified functional requirement models (Kelly and Tolvanen 2008; Stahl et al. 2006). As before, we do not claim that our modeling notations or the code generation aspect as such is new. Rather, our theoretically based focus on evolvability at both levels differs from earlier work. Therefore, it is important to mention that the resulting prototypes and applications can vary according to multiple evolvability dimensions:

– *domain model*: data elements or attributes, links or cardinalitities can be added or changed;
– *technology*: the preferred technology frameworks to implement the elements can be changed (e.g., a different database or user interface technology might be chosen);
– *plug-in code*: additional plug-in code or a better implementation of an existing functionality (plug-in code) can be required (e.g., an improved calculation algorithm);
– *element structure (design patterns)*: the element structures can change as new features are embedded (e.g., a better way of handling a cross-cutting concern).

Consequently, our approach enables the tendency towards agile approaches (e.g., DevOps) as we facilitate development by short iterative cycles in tight cooperation with end users. We also largely align with the idea of (architecture-centric) Model-Driven Software Development (MDSD) (Stahl et al. 2006), in which a model is used to generate large parts of the code. While these approaches focus mainly on the agility of the software development process, we try to complement them with products which are evolvable as well.

5 Evaluation

Our approach was iteratively refined over the course of 2 years and was mainly evaluated by means of frequent qualitative and informal feedback conversations in the context of real-life projects (differing in scope, goal and related industry) and trainings given to practitioners and students. Mentioned benefits included the ability to create more complete data models (due to the suggestions and complexity reduction offered by the different entity dimensions), a more manageable (adaptable) processing model and an increased integration between data and processing aspects. Also awareness of agility issues at the organizational level and the ability to rapidly show end users working prototypes (which allows easier communication) were regularly mentioned. Modeling NS flows on core data entities was systematically indicated as a profoundly new but positive experience for the participants. While the above mostly relates to syntactic and semantic model quality properties (Krogstie 2016), the mentioned challenges were often related to other quality properties. For instance, regarding deontic quality, Fig. 4 was considered difficult to construct in realistic situations given its (visual) complexity. Regarding pragmatic quality, the fine-grained processing models were considered as potentially difficult to understand by non-technical end users. And, regarding the physical quality, the supporting tool was sometimes experienced as error-prone when used inattentively (e.g., due to typos).

Based on this feedback, in order to get a more systematic insight, a supplementary survey (see the Appendix) with a set of statements (to be rated on a 7 point likert scale) was distributed at the end of the second year to a set of practitioners (analysts/developers) who received a 2-day training on our modeling approach (after having attended a 1-day session on NS fundamentals) and students who followed a 3-h lecture and brief follow-up session (as they were already familiar with NS at the software level). The practitioners familiarized themselves with the supporting tool via in-class exercises, whereas the students acquainted themselves via self-study. At the end, the practitioners were asked to construct an NS application within their own working context whereas the students were given an assignment to model and build an NS application for a fictitious bank. All participants were able to produce functional requirements of an NS application and build a corresponding prototype which was based on our approach, without significant problems. The statements in the survey were primarily aimed towards discovering the confidence of the respondents of being able to identify the suitable set of data, flow and task elements, and the degree to which our approach facilitated some of the benefits mentioned above. Only the responses of finalized surveys were taken into account. Moreover, regarding the students, we only considered responses coming from students enrolled within the main MIS program who were present during the lecture and follow-up session. This led to 13 valid student responses and 6 practitioner responses. Table 1 provides some descriptive statistics regarding the most pertinent evaluation questions (i.e., the median and percentage of answers with a score equal to or higher than 5, indicating that the respondent agrees to some extent with the statement).

Overall, the survey results seem to align with the findings from the informal feedback. However, interesting additional insights emerge. First, based on questions 1A, 2A and 3A, it is clear that the practitioners were more confident in their ability to identify the required data, task and flow elements and envision their interaction. This is not surprising given their practical experience (which the students lacked) and the fact that several non-evolvability related analysis challenges (e.g.,"how generally applicable should my model be?") are not targeted by our approach. Questions 1B and 2B suggest that about two thirds of the practitioners agreed that our approach (when compared to others) enables more manageable processing models and more complete data models. This latter benefit was rated even higher by the students, which might indicate that the approach might perhaps compensate to a certain extent their lacking experience in that regard. It was surprising to observe from questions 3A and 3B that only a minority of the students considered our approach to offer a (more) clear integration between data and processing requirements (although they regularly mentioned this during informal feedback moments and their assignments exhibited a tight integration between both), which is in contrast with the practitioners responses. On the one hand, it could be that our discussion of this part was not adequate enough for

Table 1 Evaluation questions

Question (shortened, see the appendix (available online via http://link.springer.com) for full questions)	Practitioners		Students	
	Median	$\geq$ 5 (%)	Median	$\geq$ 5 (%)
1A: I am able to identify the data elements	6	100	5	62
1B: I am able to create more complete data models	5.5	67	5	92
2A: I am able to identify the task and flow elements	6	83	5	54
2B: I create a more manageable processing model	5	67	5	69
3A: I am able to comprehend the data and processing interaction	5	67	4	46
3B: I am able to create models with a better integration between data and processing	5	67	4	46
4: I have a better insight regarding the agility of analysis models	5	67	5	77

students with little experience. On the other hand, a possible explanation could be that our approach automatically "forces" analysts to integrate both data and processing perspectives and that this might not always stand out for students having little real-life experience with integration problems. The survey results confirmed that, in general, respondents indicated to have gained a better insight regarding the agility of analysis models. Whereas the end assignment for practitioners was limited to 3 hours, the student assignment was a multiweek project in which they were expected to iteratively identify about 30 data elements (and associated tasks and flows). Here, the average time reported to come up with a prototype, once being familiarized with the tool, amounted to 16 hours.

It should be stressed that our evaluation is only tentative and has several important limitations. That is, our evaluation was primarily based on informal feedback and the number of respondents in the survey was limited. In particular, several validity threats may be present due to a possible response bias (e.g., instructor-pleasing behavior during informal feedback, the students filling in the evaluation form may have had a significantly more positive or negative experience with our method) or the fact that we asked respondents to (subjectively) compare their own performance to assignments in the past (when they were less experienced). As a consequence, additional evaluation (e.g., containing more respondents) which could confirm or refine our initial findings in future research is deemed appropriate.

6 Conclusions

NS aims to create evolvable software systems by systematically reusing a set of NS elements, proven to be free of combinatorial effects. This paper has proposed a modeling approach to identify these elements and discussed how these models can be used as input for a supporting tool to generate NS applications. This paper contributes to theory by illustrating the feasibility of transforming actual functional requirements into an enterprise model in terms of NS elements. We reflected upon the implications of performing enterprise modeling in compliance with NS theorems and how simple yet highly integrated models can be used to generate advanced software applications. To practitioners, this paper offers initial guidance on how to perform enterprise modeling in an NS compliant way within actual projects. The ability to use these models as direct input for a supporting tool also entails important practical implications for the development of NS software (e.g., the ability to easily generate prototypes). Future research will be mainly directed towards a more extensive evaluation of our approach and the creation of a tool to automatically transform a graphical domain model into the specifications required for the supporting tool. Also, as the resulting software applications are based on NS elements (which are designed to exhibit evolvability) and the enterprise modeling is performed while taking the NS theorems into account, one might expect an improvement of the overall agility of the adopting company. Therefore, studies validating this actual enterprise evolvability resulting from our approach within a long time perspective are deemed interesting.

References

Agarwal R, Tiwana A (2015) Editorial - evolvable systems: through the looking glass of is. Inf Syst Res 26(3):473–479

Becker J, Rosemann M, von Uthmann C (2000) Guidelines of business process modeling. In: van der Aalst WMP, Desel J, Oberweis A (eds) Business process management, models, techniques, and empirical studies, vol 1806. Lecture notes in computer science. Springer, Heidelberg, pp 30–49

Booch G, Rumbaugh J, Jacobson I (1999) Unified modeling language, the user guide. Addisson Wesley, Pearson

Chen PPS (1976) The entity-relationship model&mdash;toward a unified view of data. ACM Trans Database Syst 1(1):9–36

Codd E (1970) A relational model of data for large shared data banks. Commun ACM 13(6):377–387

de Cesare S, Serrano A (2006) Collaborative modeling using uml and business process simulation. In: Proceedings of the 39th annual HICSS conference

De Bruyn P (2014) Generalizing normalized systems theory: towards a foundational theory for enterprise engineering. PhD thesis, University of Antwerp

De Bruyn P, Huysmans P, Mannaert H (2016) Tailoring an analysis approach for developing evolvable software systems: experiences from three case studies. In: Proceedings of the 18th conference on business informatics, pp 208–217

Eessaar E (2016) The database normalization theory and the theory of normalized systems: finding a common ground. Baltic J Mod Comput 1:5–33

Eichelberger H, Eldogan Y, Schmid K (2009) A comprehensive survey of uml compliance in current modemodel tools. Softw eng 143:39–50

Evans E (2003) Domain-driven design: taking complexity in the heart of software. Addison-Wesly, Boston

Evéquoz F, Sterren C (2011) Waiting for the miracle: Comparative analysis of twelve business process management systems regarding the support of BPMN 2.00 palette and export. Tech. rep., University of Applied Sciences Western Switzerland

Halpin T (2001) Information modeling and relational databases. Elsevier, Amsterdam

Hay DC (1996) Data Model patterns: conventions of thought. Dorset House, New York

Hruby P (2006) Model-driven design using business patterns. Springer, Heidelberg

Jackson M (1975) Principles of program design. Academic Press, Cambridge

Kappelman M, Eand McLean V, Johnson Gerhart N (2014) The 2014 SIM IT key issues and trends study. MIS Q Exec 13(4):237–263

Keller G, Nüttgens M, Scheer A (1992) Semantische Prozessmodellierung auf der Grundlage Ereignisgesteuerter Prozessketten

(EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik (89)

Kelly S, Tolvanen JP (2008) Domain-specific modelling: enabling full code generation. Wiley, New jersey

Krogstie J (2016) Quality in business process modeling. Springer, Heidelberg

Larman C (1997) Applying UML and patterns. Prentice Hall, New jersey

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

van der Linden D, De Sitter G, Verbelen T, Devriendt C, Helsen J (2017) Towards an evolvable data management system for wind turbines. Comput Stand Interface 51:87–94

Mannaert H, Verelst V, Ven K (2011) The transformation of requirements into software primitives: studying evolvability based on systems theoretic stability. Sci Progr 76(12):1210–1222 (Special Issue on Software Evolution, Adaptability and Variability)

Mannaert H, Verelst J, Ven K (2012) Towards evolvable software architectures based on systems theoretic stability. Softwa Pract Exp 42(1):89–116

Mannaert H, Verelst J, De Bruyn P (2016) Normalized systems theory: from foundations for evolvable software toward a general theory for evolvable design. Koppa

Mendling J, Reijers H, van der Aalst W (2010) Seven process modeling guidelines (7pmg). Inf Softw Technol 52(2):127–136

Moody D (2003) The method evaluation model: a theoretical model for validating information systems design methods. In: Ecis 2003 proceedings

Moreno-Montes de Oca I, Snoeck M (2014) Pragmatic guidelines for business process modeling. Technical Report, Leuven

Reijers H, Mendling J (2008) Modularity in process models: review and effects. In: Dumas M, Reichert M, Shan MC (eds) Business process management, vol 5240. Lecture notes in computer science. Springer, Heidelberg, pp 20–35

Sammon D, McNulty J, Sugrue A (2016) Tasc2c: desdesign a data driven business process. J Decis Syst 25(S1):639–646

Sanz J (2011) Entity-centric operations modeling for business process management: a multidisciplinary review of the state-of-the-art. In: Proceedings of the 6th IEEE international symposium on service oriented system engineering (sose), pp 152–163

Scheer A (1998) Business process engineering: reference models for industrial enterprises. Springer, Heidelberg

Silverston L (2001) The data model resource book: a library of universal data models for all enterprises, vol 1. John Wiley, New Jersey

Stahl T, Völter M, Bettin J, Haase A, Helsen S (2006) Model-driven software development. Wiley, New Jersey

Van Nuffel D (2011) Towards designing modular and evolvable business processes. PhD thesis, University of Antwerp

Vergidis K, Tiwari A, Majeed B, Roy R (2007) Optimisation of business process designs: an algorithmic approach with multiple objectives. Int J Prod Econ 109(1):105–121

Xiao L, Zheng L (2012) Business process design: process comparison and integration. Inf Syst Front 14(2):363–374

Yourdon E, Constantine L (1979) Structured design: fundamentals of a discipline of computer program and systems design. Prentice Hall, New Jersey