Identifying Combinatorial Effects in Requirements Engineering

Jan Verelst[1], Alberto Rodrigues Silva[2], Herwig Mannaert[1], David Almeida Ferreira[2], Philip Huysmans[1]

jan.verelst@ua.ac.be, alberto.silva@inesc-id.pt, herwig.mannaert@ua.ac.be, david.ferreira@inesc-id.pt, philip.huysmans@ua.ac.be

[1] Normalized Systems Institute
Department of Management Information Systems
University of Antwerp
Antwerp, Belgium

[2] Department of Computer Science and Engineering
IST & INESC-ID
Lisbon, Portugal

Abstract. There are several best practices and proposals that help to design and develop software systems immune (to some extent) to combinatorial effects as these systems evolve. Normalized Systems theory, considered at the software architecture level, is one of such proposals. However, at the requirements engineering (RE)-level, little research has been done regarding this issue. This paper discusses examples related with this problem considering two distinct RE abstract levels, namely at the business and system levels. The examples provided follow the notations and techniques typical used to model the software system at such levels, namely DEMO/EO, BPMN, and UML (Use Cases and Class diagrams). The analysis of these examples suggests that combinatorial effects can be easily found at these different levels. This paper also proposes a research agenda to further investigate this matter in terms of the effects of combinatorial effects, and envisions the mechanisms and solutions for dealing with them. It is suggested that an artifact-based, domain-specific approach is best suited to achieve highly agile enterprises and RE-processes in the future.

Keywords. Requirement engineering (RE); requirements specifications; combinatorial effects (CE); normalized systems.

1 Introduction

A software requirements specification is a document that describes multiple technical concerns of a software system [1,2]. A requirements specification is used throughout different stages of the project life-cycle, namely to help sharing the system vision among the main stakeholders, as well as to facilitate their communication, the overall project management, and system development processes. A good requirements specification provides several benefits, namely [7,19-24]: establishes the basis for

agreement between the customers and the suppliers on what the system is expected to do; reduces development efforts; provides a basis for estimating costs and schedules; provides a baseline for verification and validation; facilitates the system deployment; and serves as a basis for future maintenance activities.

Over the past two decades, it has become clear that organizations are increasingly facing more volatile environments. However, there are many indicators that organizations typically find difficult to cope with these changes in terms of their information systems. For example, the high percentage of challenged or even failed IT-projects clearly illustrates this problem [28]. For better describing this situation, some authors have even coined the term "software crisis". However, *change* does not only affect software, more specifically information systems. The respective requirements specifications are affected as well. Moreover, requirements specifications are the earliest documents in the systems development life cycle, and thus one of the first artifacts to be affected by change. Therefore, requirements specifications should be capable of dealing with change, namely taking preventive measures in terms of their structure and content to avoid such changes from causing a ripple effect at subsequent software development phases, such as software design and implementation.

Normalized Systems (NS) theory is especially concerned in studying the behavior of modular structures, such as software architectures, under change [4,5]. From a systems theoretic perspective, this theory has shown that evolvability and flexibility are largely determined by the presence of combinatorial effects (CE). Such CEs can be regarded as a kind of coupling and, more specifically, a ripple effect that is independent from aspects such as programming languages, systems development methodologies or frameworks used. Furthermore, CEs exhibit a highly harmful characteristic: these effects grow as the modular structure grows larger, which commonly occurs in practice over time. According to these empirical observations, the behavior of CE correlates with Lehman's law of increasing complexity, which states that, as maintenance is performed on a software system, its structure degrades and becomes more complex, thus making it inflexible [6]. This way, the existence of CEs explains why and how Lehman's law occurs. Furthermore, NS theory suggests that studying evolvability, as well controlling CE, is a highly complex endeavor, as CEs can occur at many levels in information systems and software architectures, from high-level effects at RE-level to very detailed effects at the implementation level. Usually, most CEs can be found at the lower level, where the large amounts of (cross-cutting) concerns make it difficult to avoid them.

In this paper, we report on our experiences focused on CEs at RE-level. Our research was motivated by several goals. Firstly, RE is a crucial discipline to be performed at the beginning of the systems development process. The existence of CEs at RE-level indicates that these modular artifacts exhibit limited evolvability, regardless of the software systems derived from them. In general, this limited evolvability is problematic because of the considerable effort involved in the RE process, as well as the impact of its main delivery in terms of the remaining phases of the software development process. Additionally, it also negatively influences the motivation and ability of requirements engineers to update their artifacts over time, which can lead to misalignments between the requirements specification and the software where these

changes were effectively applied. Given that requirements specifications are often the basis for complementary technical documentation about the information system, as well as part of legal documents surrounding the corresponding project (including RFP or Project Contracts), several problems can result from CEs at RE-level.

Secondly, the concept of evolvability at RE-level is often overlooked. For example, in object-oriented (OO) literature, it is sometimes assumed that RE is substantially based on anthropomorphism, in the sense that making models about the problem space mainly consists in passively identifying real world objects. This approach has been argued by authors such as Simsion, who claim that data modeling should be considered more a design activity than an analysis activity [27]. The presence of CEs and coupling favors the latter perspective, and suggests that the RE-process entails many more issues than following a passive, analysis-like identification of objects in the real world. Indeed, NS theory suggests that studying evolvability is a complex, multi-level approach that, in turn, suggests different and more complex approaches to study the evolvability at RE-level are still needed.

The paper is organized as follows. Section 2 introduces the NS theory and the constructs and models commonly used in requirements specifications. Next, Section 3 provides examples of CEs using different notations and techniques, such as DEMO, BPMN, and UML diagrams. In Section 4, we argue that these examples illustrate the need for a systematic research agenda on the identification and control of CEs at RE-level. Finally, Section 5 presents our conclusions.

2 Background

This section introduces the NS theory and the constructs and models usually used at the requirements specifications level.

2.1 Normalized Systems Theory

Normalized Systems (NS) theory studies how modular structures behave under change [4,5]. Initially, this theory was developed by studying change and evolvability at the software architecture level, by applying concepts such as stability and entropy to the study of the modular structure of the software architecture. Considering the application of systems theoretic stability to software architecture, *stability* implies that a bounded input function should result in bounded output values, even as $T \rightarrow \infty$. In software architecture, this means that a bounded set of changes should result in a bounded amount of changes or impacts to the system, even for $T \rightarrow \infty$. The concept of stability warrants that the amount of impacts caused by a change cannot be related to the size of the system and, therefore, needs to remain constant over time as the system grows. In other words, stability demands that the impact of a change is only dependent on the nature of the change itself. If the amount of impacts is related to the size of the system, a *combinatorial effect* (CE) occurs.

Research has shown that it is very difficult to prevent CEs when designing software architectures. More specifically, it has been proven that CEs are introduced each time one of four theorems is violated. The first theorem, *separation of concerns* , implies that every change driver or concern should be separated from other concerns. Applying this principle prescribes that each module can only contain one submodular task (which is defined as a change driver), but also that the implicit workflow should be separated from functional submodular tasks. The second theorem, *data version transparency* , implies that data should be communicated in a version transparent way between components. This requires that this data can be changed (e.g., additional data can be sent between components), without having an impact on the components and their interfaces. The third theorem, *action version transparency* , implies that a component can be upgraded without impacting the calling components. The fourth theorem, *separation of states*, implies that actions or steps in a workflow should be separated from each other in time by keeping state after every action or step. This suggests an asynchronous and stateful way of calling other components.

The proofs of the theorems show that unless every theorem is adhered to at all times during maintenance, the number of CEs will increase, making the software more complex and less maintainable. This can only be avoided when software is developed in a highly controlled way, ensuring that none of these principles are violated at any point in the development process during development or maintenance, which is quite difficult to achieve in practice. A modular structure that is free from CE, is called a Normalized System (NS). In order to achieve this, CEs should not be present at compile time, deployment time, and run time in modular structures. Furthermore, it has been shown that software architectures without CEs can be built by constructing them as a set of instantiations of highly structured and loosely coupled design patterns (called *elements*), which provide the core functionality of information systems and are proven to be free of CE.

This approach allows considering these software patterns as reusable building blocks, which can be aggregated using a mechanism called *expansion* to build information systems based on these building blocks without introducing CE. This contributes to realizing the vision of Doug McIlroy, who hoped for a future for software engineering in which software would be assembled instead of programmed. It is important to note that such assembly requires modules which are purposefully designed to prevent CE. Only when the absence of CEs in every pattern has been confirmed, it is possible to reuse these patterns without consulting their internal construction. Putting it in other words, only then can they be regarded as black boxes for usage in information systems. The theorems and patterns are described in terms of modular structures, which are independent of a given programming language or paradigm. As a result, these theorems and patterns have a wide applicability. More importantly, this shows that, in order to identify CE, and prescribe guidelines to prevent them, a modular structure in the domain under investigation needs to be made explicit, and the reuse of the modules in a black-box way should be confirmed.

2.2 Constructs and Models in RE

Requirements specifications define a somehow rigorous set of statements that help sharing a common vision between business stakeholders and the development team, and facilitates the communication, negotiation and managing efforts among all involved stakeholders. In general, requirements are specified in natural language due to their higher expressiveness and ease of use [7]. However, the usage of unconstrained natural languages often presents some drawbacks such as ambiguity, inconsistency and incompleteness. To mitigate some of these problems, specifications in natural language are typically complemented by some sort of controlled or semi-formal language − usually graphical languages such as UML [8], SysML [9], i* [10] or KAOS [11] −, which address different abstraction levels and concerns. Usually requirements engineers consider two distinct abstraction levels when organizing and specifying requirements: business level and system level. At the business level they define the enterprise and business context, and also the purpose and general goals of the system; while at the system level they have to further detail the concrete technical requirements of the system.

The constructs considered *at business level* are commonly the *terminology*, the business *goals* that the system should satisfy, and the *stakeholders* that are the sources of these goals and requirements, but also *business processes* and *business use cases*. There are in the community some languages that address the design of *goal-oriented models*, namely i* and KAOS. There are also other approaches to describe the system scope at this level, namely UML [8,12], BPMN [13], and RUP business modeling [14]. Additionally, depending on the size and complexity of the systems in consideration, enterprise engineering (EE) approaches can also be adopted at this level, for example using languages such as DEMO [15] or Archimate [35].

On the other hand, the main models considered in requirements specifications *at the system level* are context models, domain models, functional requirements models, and quality-attributes models. *Context models* use constructs such as *system, subsystems, components, nodes, external actors*, and respective relationships such as communication, interoperation, decomposition or deployment. Some of the visual languages that can be used to represent context models are SysML Block diagrams, UML Deployment diagrams, Data Flow Diagrams (DFD) at the context level [18], or even informal Block diagrams.

Domain models use constructs like *entities* or *classes*, and respective relationships such as associations and generalizations, and help to capture the key concepts or information resources underlying the system. The common graphical languages used to produce domain models are ER (Entity-Relationship) diagrams [18] or UML Class diagrams.

Functional requirements models use constructs such as *actors*, *functional requirements*, *use cases, scenarios* or *user stories*. There are different approaches to specify functional requirements. Most of these approaches recommend the use of textual specifications, written according to linguistic patterns properly enriched with predefined metadata and classifiers, such as priority and risk levels, authors, or creation dates. Other approaches recommend simple graphical representations such as UML

Use Case diagrams or SysML Requirements diagrams. Yet, others recommend hybrid approaches by combining textual and graphical descriptions.

The concept of *non-functional requirements* (NFR) corresponds to high-level business constraints, technical constraints, and quality attributes [16,17]. Usually *business constraints* (e.g., a constraint related to the budget or the schedule of the project) are business level NFR but not included in requirements specifications because they used to be defined in other documents such as Project Charter and Project Plan documents. On the other hand, *technical requirements* (e.g., a constraint related the use of a specific development tool, the use of a particular database management system, or the adoption of a particular software development process) and *quality attributes* are considered system level NFR. *Quality-attribute models* use constructs like *qualities, metrics* and *utility values* to specify transversal properties of the system, such as maintainability, usability, performance, security, privacy or scalability. There are also some approaches to specify these requirements, namely the quality-attributes scenarios [17], or simple lists of quality-attributes [21]. Although quality-attributes are not difficult to be identified, they are hard to quantify in a verifiable manner. Since they can have a huge impact on the overall cost of the solution, they must be properly considered at the software architecture level [17].

3 Identifying Combinatorial Effects at RE Level

In this section we provide some illustrative examples of CEs that exist at requirements specifications based on the discussion of some notations and techniques commonly used, namely based on DEMO/EO, BPMN, and UML (in particular based on its Use Cases and Class diagrams). However, we start by discussing whether CEs can exist at the enterprise level (real world level), *irrespective* from these RE techniques.

Table 1. Analysis of Languages used in RE regarding Modularity and Combinatorial Effects

Abstract Levels	Approaches	Languages	Concepts	Relationships	Decomposition
from Real World …					
Business	Enterprise Ontology	DEMO	Service, Transaction, Act, Actor Role	Communication, Coordination, Production	Services compounded of transactions, transactions compounded of acts
	Business Processes	BPMN	Process, Resource	Control flow, Data flow	Process decomposition
System	OO System Analysis	UML Class Diagrams	Class	Association, Generalization	Class aggregation and composition
		UML Use Cases Diagrams	Use Case, Actor	Include, Extend	-
… into Software Systems					

Table 1 summarizes the key aspects discussed below. As referred in section 2.2, requirements specifications can be defined at two distinct and complementary abstraction levels: business and system levels. The models produced at these abstraction

levels can be somehow classified as those used in MDE (Model Driven Engineering) paradigm [36]. For example, considering the OMG MDA (Model Driven Architecture) approach, they can be classified, respectively, as Computational Independent Models (CIMs) and Platform Independent Model (PIMs). As it is expected, there are not Platform Specific Models (PSMs) defined at the RE level. We start by discussing the use of DEMO/EO and BPMN at the business level and then the UML (class and use cases diagrams) at system level. However, we do understand that because UML and (in somehow) DEMO are general-purpose modeling languages they could be used at both abstract levels.

3.1 from the Real World…

In information systems literature, it is commonly assumed (at least to a certain extent) that the information system should mirror the real world [26,27], which is also suggested by the concept of *anthropomorphism* that is frequently cited in object oriented literature. Together with communications theory-based approaches, such as DEMO, this would suggest that the real world is first and foremost an area of human behavior, which should therefore not predominantly be studied by theories based on computer science and/or automation. We agree with this point of view. Nevertheless, in modern society, human behavior increasingly takes place in highly structured, process-based contexts. Therefore, we argue that it is relevant to study these aspects of reality based on concepts such as *modularity*, while at the same time making an *abstraction* from purely human and communication aspects.

Therefore, an initial area for applying NS theory is the real world being mirrored. In other words, the first possibility is to investigate whether the real world itself consists of modular structures that are inherently unstable from a system theoretic perspective. To illustrate this, we take a simple example of a completely manual information system (not automated at all) at a university where student marks have to be rounded. Suppose the university has the policy of rounding exam marks "to the nearest integer". The university has the option to ask all professors to perform this rounding (option 1), but also to ask the administrative exam secretariat to perform this duty (option 2). Suppose now that the university policy changes: following option 1, the change impacts the number of professors that have to be notified to change their behavior, which is related to the size of the university, thus emphasizing a CE. In option 2, only one actor needs to be notified (the exam secretariat), implying that just 1 (or a few) physical person(s) have to be notified, which is largely or fully independent of the size of the organization. Therefore, option 2 has no (or only a negligible) CE. We stress that this example focuses on a system with no automated processing involved. Therefore, there is no combinatorial effect in the automation, but in 'the real world'.

A second example of a CE in the real world concerns the traditional *versus* virtual mail distribution. In certain organizations, most employees are entitled to write (physical) letters to external stakeholders. However, the logo's and letterheads of organizations are these days frequently changed, resulting in different paper and envelopes

being used, and in this scenario, impacts every (secretary of) letter authors. This impact is dependent on the size of the organization, thus emphasizing another CE. Increasingly, organization are virtualizing their letters, by having authors send electronic versions of their letters to an internal or external mail center, who prints them, puts them in envelopes and dispatches them. In this second scenario, only one part of the organization is affected by the change of a company logo and letterhead, and therefore, no CE is present, or only an inconsequential one.

Both examples illustrate the existence of CE, without or prior to the use of RE techniques or notations, suggesting that they exist in the "Real World". Such CEs are (in a certain sense) outside the scope of the requirements engineer, as it is up to the business stakeholders to decide how to structure their organization and business processes.

3.2 DEMO/EO

An approach to RE is to start from enterprise models in order to give a high-level view of the business, and technical context of the system-of-interest. Among other alternatives, DEMO [15] have been used to support this goal, as well as a starting point for deriving use cases that describe the system functionality [29]. This is interesting for our approach, since DEMO models may be considered to be appropriate for analyzing CEs for the following reasons.

First, DEMO claims to create *constructional models*, instead of functional models. Constructional models represent the actual components of which a system consists. In contrast, functional models do not represent system components, but describe instead how a stakeholder uses the system. Possibly, this distinction explains why in Section 3.4 no CE could be identified: functional models do not consider the (modular) structure of a system, which was considered to be a prerequisite for identifying CEs in Section 2. In contrast, modular discussions based on DEMO models have already been described: for example Op't Land [30] argues that cohesion and coupling between actors in DEMO models can be used to decide whether or not to keep organizational actors together when splitting organizations.

Second, DEMO explicitly considers organizational building blocks, and prescribes rules for their aggregation. Acts are considered to be the basic building blocks (i.e., atoms), which are combined to create transactions (i.e., molecules). In order to deliver services to the environment, collections of transactions are invoked (i.e., fibers). The composition axiom structures how transactions can be interrelated. Transactions are either (1) initiated externally, (2) enclosed, or (3) self-initiated. Therefore, the aggregation of transactions needs to occur in certain ways. NS theory shows that CEs are often introduced when aggregating such constructs, and that prescriptive guidelines are required to show how building blocks can be aggregated without CE. Because DEMO models have clearly defined building blocks and aggregation guidelines, an analysis of the attention given to CEs on this level could be feasible. To the best of our knowledge, it has not been researched yet whether eliminating CEs has been taken into account in the DEMO guidelines.

Third, it is at least remarkable that certain concepts from NS theory are similar to EO [31]. For example, consider the *separation of states* theorem. It states that "the calling of an action entity by another action entity needs to exhibit state keeping in normalized systems" [4]. Therefore, it prescribes how action elements can interact. This impacts, for example, the workflow element, which aggregates action elements. A workflow can reach different states by performing state transitions. A state transition is realized by an action element. The successful completion of that action element results in a defined life cycle state. The workflow specification determines which state transitions can be made. Similarly, the state of a transaction in EO is determined by the successful performance of acts. The result of such an act results in the creation of a defined fact. Despite the different terminology, a clear resemblance between NS and EO emerges: state keeping is enforced in NS theory by defining states, and in EO by creating facts. These NS states are the result of executing actions, whereas the EO facts are the result of executing acts. The set of actions that can be performed is determined by state transitions in NS, and occurrence laws in EO. While we do not claim the adherence of DEMO models to the *separation of states* theorem, it is remarkable that such similar concepts are implicitly used, specially considering the different theoretical background of both approaches (i.e., language-action perspective and systems theoretic stability, respectively).

Notwithstanding these arguments, it should be noted that many real world aspects cannot be represented in DEMO models, since they are implementation-independent. Consider for example the "round to the nearest integer" (example described above). Using DEMO models, no difference between the two situations could be determined: at most, the rounding is an action rule for a certain execution act. Who applies this action rule is not modeled: the person fulfilling the executor actor role for this transaction (e.g., the examiner) can apply it, or it can be delegated (to the exam secretariat). Therefore, certain CEs of the implementation in the real world will not be visible in DEMO models.

3.3 BPMN

BPMN [13], like other notations (e.g., UML Activity diagrams), allows modeling business processes and, hence modeling the business context of the system in consideration. BPMN provides constructs such as process, task, role, resources, and so on, and also relationships such as control- and data-flows. In BPMN, processes and tasks can be considered and analyzed as modular structures. Research has already identified CEs in business processes, and provided guidelines to prevent them [32]. As such, changes to a certain process will only need to be applied in a single process model, instead of in every model where the functionality of that process is needed.

For example, consider a payment process. A payment process constitutes a different concern than the business process that handles, for instance, a purchase order. Based on the *separation of concerns* principle, this functionality should therefore be isolated in a dedicated process. If the payment functionality is modeled in every business process requiring a payment, each of these processes would have to be able to capture every possible change in the payment functionality. For example, when cash

payments are no longer allowed, or validating an e-banking transfer with a first-time customer. However, if the payment concern is isolated in its dedicated business process, only the dedicated payment process needs to be changed. All the fault handling regarding transactions is included in this process. As a result, a reusable process can be modeled. Any business process requiring a payment, can request an execution of the payment process.

Based on the *separation of concerns* principle, a set of 25 guidelines has been proposed to eliminate CEs in business process models. Each guideline starts from the identification of a possible CE, and prescribes a solution to prevent that CE. As such, these guidelines are less general than the NS theorems. Rather, these principles apply the NS theorems on the business process model level. Although requirements are not expressed using process models, these guidelines illustrate how CEs can be identified and prevented at the business level of requirements specifications.

3.4 UML Use Cases

Use cases are highly popular, detailed and semi-formal descriptions of functional requirements. By far, UML Use Cases diagrams are the most popular graphical representations of the system from its functional point of view. These diagrams depict the actors (i.e., end-users and external systems) that interact with the system through a well-defined number of use cases. Use Cases are related among themselves through include or extend relationships. In the end, use cases are described textually, and are therefore typically situated close to the real world-level.

On one hand, use cases do have some modular characteristics, namely: (1) the name of the use case can be considered a primitive form of interface; (2) pre- and post-conditions can also be considered to delineate the functionality of the use case, and therefore be considered part of the interface (more specifically, another use case can treat this use case as a black box, providing the functionality described in the post conditions); and (3) the workflow of the use case can be considered the content of the module.

To a certain extent, this allows the identification of potential CE. For example, the principle of *separation of concerns* can be applied to (groups of) steps in the workflow. Typically, this principle is violated when several Use Cases describe the same functionality, or even terminology in a redundant way. If such a redundancy does not exist "in the real world", but does exist in the Use Case, it is a CE at the Use Case-level, caused by the text-based constructs used in Use Cases. Other constructs may help to prevent such CE: for example, *tagging* parts of a workflow or individual user interface requirements could be used, to provide a hypertext-like structure, which supports the identification of the impact of certain CE, and perhaps even the reduction of the number of impacts. Even though hypertext and tagging have limitations in terms of coupling in modular structures, they at least provide a better structure than plain text to judge the presence of CE.

However, textual descriptions have severe limitations in terms of CE. Use Cases are usually too underspecified to allow thorough identification of CE. For example, *action version transparency*, *data version transparency*, and *separation of state* theo-

rems can be applied to the module interface (describing when Use Cases call each other). In Use Cases, however, this is difficult to judge because no interface parameters are detailed. Also, Use Cases give virtually no guidance as to which concerns should be separated, and therefore they are prone to scatter certain concerns over the entire document. In turn, this can lead to mixing functional with non-functional concerns, as well as mixing several non-functional concerns. For example, Use Case documents could contain non-functional user interface details in many or every Use Case (describing functional concerns). A change in the user interface requirements may then require a very large number of updates across the entire document.

This under-specification and lack of guidance is typically pointed out as one of the criticisms of EE (Enterprise Engineering) researchers, regarding Use Cases. Indeed, identifying CEs can be done in a more precise way in EE-approaches, such as DEMO.

3.5 UML Object-Oriented Domain Models

As mentioned in the previous section, domain models capture the key concepts of the system of interest. UML class models are the most popular notation for such domain models. Such models depict the classes and their relationships, such as associations and generalizations. Each class can both define data (attributes) and functions (methods) properties.

Concerning data, redundant definition of attributes are well-documented examples of CE, and the application of Codd's normalization rules [34] eliminates many of them. Concerning functions, the use of atomic data types in the interface of a method is a common violation of data version transparency. The CEs becomes clear when changing the definition of the attribute, which subsequently has to be applied to all redundant instances of the attribute. On the other hand, concerning the relationships between classes, the use of "sync pipelines" is an example of a violation of separation of state. This refers to a typical style in OO analysis, design and programming where method A calls method B, which calls method C, which calls method Z… while method A is still waiting for a return value from B. Similarly, X could call Y, who calls Z. In this case, the addition of one new error state in Z, would impact every calling method, i.e. both Y (and probably X), and C (and probably A and B).

These CEs are very similar to those in OO programming, which have been documented in [4].

4 Discussion

This discussion reflects our experience in the field both in practice and research. On one hand, we have more than 10 years' experience specifying as well assessing and auditing complex information systems based on different types of requirements specifications and related documentation. On the other hand, we have also researched in areas such as software architectures and system design (e.g., the Normalized Systems theory and its application [4,5]), requirements specification languages (e.g., the ProjectIT-RSL [23] or RSLingo approaches [24]), and the alignment between RE and

MDE fields [25]). The scope and contribution of this paper reflects this blended experience.

The examples of CEs mentioned in the previous section are relatively straightforward, but they are sufficient to illustrate the omnipresence of instabilities in a domain that is sometimes considered to be about "identification of objects in the real world". Indeed, these examples illustrate that both the real world (enterprise itself and respective information systems) and the "mechanisms and tools" we use to model them (e.g., classes or informational entities, use cases, processes and workflows, enterprise ontologies and communication acts) contain these instabilities. All of these CEs will exhibit Lehman-like symptoms. Initially, when the system is small, they would probably not be problematic, but over time their effects would grow and slowly but surely increase the rigidity of requirements models and specifications (which are sometimes used as the technical documentation of the information system, or a component in a legal contract concerning the system).

As summarized in Table 1, the examples above illustrate the existence of CEs at different abstraction levels, from the "real world" to enterprise-level and business processes descriptions (such as DEMO and BPMN), to object-oriented UML diagrams that have similar constructs as the implementation levels of information systems. This suggests that the RE- and systems development process consists of bridging a set of functional/constructive gaps, where every constructive level realizes the functional requirements of the functional level above using its own constructs.

At every level of this set of functional/constructive gaps, a certain amount of control of CEs should be striven for. On the one hand, it is clear that it is advantageous to eliminate the CE, and that is what we advocate at the software level: maximal elimination of CE. However, we explicitly mention that this is not necessarily the case at higher levels. For example, at the enterprise level, it seems quite probable that a certain level of CEs can be tolerated (for example, an organization may decide to keep using physical letter distribution over a virtual mail system), but in any case this decision should then be taken in a conscious way. At the level of the RE-techniques, it seems more certain that one should strive for full control of the extra CEs that are incurred. For example, the coupling in text-based requirements specifications' should be investigated for additional CE, as well as coupling CEs in BPMN models [32].

It should be remarked that the examples shown above are relatively straightforward, and that an experienced RE-practitioner is probably currently able to deal with most of these CEs in a heuristic way (based on experience). However, at the software level with its high number of concerns and correspondingly complex modular structures, heuristics have shown to be insufficient to control the large number of highly complex CEs that are responsible for the symptoms of Lehman's law. The enterprise and its supporting information systems are widely assumed to become even more complex in the future. This perspective may imply enterprises becoming larger in terms of amounts of different products, markets and human resources, but also relatively small enterprises can be faced with very high levels of complexity as they are part of multi-actor value networks that grow in size and complexity.

As this process of complexity increase takes place, heuristics applied by individual members of a RE-team will increasingly fall short in controlling coupling at the RE-

level, and the need for a systematic approach to dealing with CEs is increasingly needed. Such a systematic approach should address minimally the following issues:

First, identification of CEs at each level, both in the constructs of the level and the models built using these constructs (also other NS-related concepts such as entropy should be considered at each level, but this is outside the scope of this paper).

Second, different mechanisms to achieve control of CE, such as the code generation or expansion mechanisms that was used at the software level [5], but perhaps manual or semi-automated mechanisms are more appropriate at higher levels.

Third, appropriate levels of control of CE, and extent to which they need to be applied in different levels of circumstances. For example, as mentioned above, possibly the higher levels should have higher tolerance-levels for CEs than more implementation-oriented levels. It is also possible that different levels are required depending on the sector the enterprise is situated in (for example, the types of changes that occur at high frequency in a sector).

The combination of these three issues, suggests that a single abstract, domain-independent approach in unlikely to achieve this ambitious goal of building the agile enterprise of the future. It is more likely that domain-dependent approaches are needed to focus fully on the subtle and complex issues surrounding coupling at all different levels in a certain sector. This is similar to classical engineering where reusable, domain-specific artifacts are constructed in sector like computer hardware design, car manufacturing, etc. Coupling is in these approaches also addressed by splitting the problem of car manufacturing in a series of sub-problems, i.e. design and manufacturing of the engine, the dashboard etc. Such a domain-dependent approach would mean that loosely coupled artifacts need to be developed in areas such as finance, accounting, transport, human resources, or in subareas such as invoicing, staffing, project management, mail distribution, payments, etc. All of these transversal subareas contain highly complex coupling issues which can be addressed by developing artifacts such as invoice lines, address validators, credit checkers etc. When these artifacts are developed using a modular structure which exhibits control of coupling issues (such as a low number of CE), they can be aggregated into higher-order structures such as an invoice. This example may be surprisingly uncomplicated, but at this point in time, there is no accurate description of the modular structure of an invoice available in the scientific literature, which is (widely) used in practice. On the contrary, invoices are currently still defined in practice in product-dependent and/or heuristic way, with no explicit study or science-based control of their modular structure.

Therefore, we believe that RE and EE would benefit from a piecemeal and inductive research agenda that is allowed by these domain-specific and problem decomposition approaches, in order to perhaps generalize to domain-independent techniques and methodologies in the future. However, we remark that this approach contrasts with the current mainstream in RE-literature, where there is a focus on large numbers relatively domain-independent modeling languages (constructs), techniques, methodologies and tools being proposed, with limited systematic study of the characteristics of the artifacts that are constructed.

5 Conclusion

In this paper we have documented our experiences in looking for CEs at different levels in the RE-process. The examples cover CEs at the RE level based on the adoption of notations and techniques such as UML classes and use cases, DEMO/EO, and BPMN models. The examples presented are relatively straightforward, but enough to show the omnipresence of such instabilities in the RE levels. As a result, we have described the need for a research agenda focusing on the systematic research into CEs and related issues at the RE domain in order to build enterprises and their information systems that are able to exhibit new levels of agility that will be required in the future.

In this way, we support the call by Dietz et al. for the area of Enterprise Engineering to be developed [33]. The amount and complexity of issues that need to be solved to achieve the next generation of truly agile enterprises both in the service and industrial sector, both in the for-profit and not-for-profit sector, is such that a scientific basis focusing on structural issues (including coupling) will be required.

References

[1] K. Pohl, Requirements Engineering: Fundamentals, Principles, and Techniques, 1st ed. Springer, 2010.

[2] I. Sommerville and P. Sawyer, Requirements Engineering: A Good Practice Guide. Wiley, 1997.

[3] T. T. Tun, T. Trew, M. Jackson, R. Laney, B. Nuseibeh, "Specifying features of an evolving software system", Software: Practice and Experience", 2009; 39(11):973-1002, doi:10.1002/spe.923.

[4] H. Mannaert, J. Verelst, Normalized Systems: Re-creating Information Technology Based on Laws for Software Evolvability, Koppa, 2009.

[5] H. Mannaert, J. Verelst, K. Ven, "Towards evolvable software architectures based on systems theoretic stability", Software Practice and Experience, Wiley, 2012.

[6] M. M. Lehman, "Programs, life cycles, and laws of software evolution", Proceedings of the IEEE Sept 1980; 68(9):1060-1076.

[7] B. Kovitz, Practical Software Requirements: Manual of Content and Style. Manning, 1998.

[8] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide. Addison-Wesley, 2005.

[9] OMG, Object Management Group, Systems Modeling Language, available at: http://www.omgsysml.org

[10] E. Yu, "Modelling Strategic Relationships for Process Reengineering", PhD thesis, University of Toronto, Canada, 1995.

[11] A. Lamsweerde, Requirements Engineering: From System Goals to UML Models to Software Specifications. Willey, 2009.

[12] N. Castela, J. Tribolet, A. R. Silva, A. Guerra, "Business Process Modeling with UML", Proceedings of the International Conference on Enterprise Information Systems, ICEIS Press, 2001.

[13] OMG, "Business process model and notation (bpmn), version 2.0," tech. rep., OMG, 2011.

[14] IBM Rational Method Composer and RUP on IBM Rational developerWorks, http://www.ibm.com/developerworks/rational/ products/rup/

[15] J.L.G. Dietz, Enterprise Ontology: Theory and Methodology, Springer, 2006.

[16] L. Chung, J. Leite, "On Non-Functional Requirements in Software Engineering", in Conceptual Modeling: Foundations and Applications, LNCS 5600, Springer, 2009.

[17] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice, 2nd edition. Addison Wesley, 2003.

[18] P. Weaver, Nick Lambrou, M. Walkley, Practical SSADM Version 4+, 2nd edition, Prentice Hall, 1998.

[19] IEEE, IEEE Std 830-1998 (Revision of IEEE Std 830-1993). IEEE Recommended Practice for Software Requirements Specifications. 1998.

[20] S. Withall, Software Requirements Patterns, 2007, Microsoft Press.

[21] S. Robertson, J. Robertson, Mastering the Requirements Process, 2nd edn. Addison-Wesley, 2006.

[22] A. Cockburn, Writing Effective Use Cases. Addison-Wesley, 2001.

[23] C. Videira, D. Ferreira, A. R. Silva, "A linguistic patterns approach for requirements specification", in Proc. 32nd Euromicro Conference on Software Engineering and Advanced Applications, IEEE Computer Society, 2006.

[24] D. Ferreira and A. R. Silva, "RSLingo: An Information Extraction Approach toward Formal Requirements Specifications", in Proc. of the 2nd Int. Workshop on Model-Driven Requirements Engineering (MoDRE 2012). IEEE Computer Society, 2012.

[25] A. R. Silva, J. Saraiva, D. Ferreira, R. Silva, C. Videira, "Integration of RE and MDE Paradigms: The ProjectIT Approach and Tools", IET Software Journal, 1(6), IET, 2007.

[26] A. Borgida, "Features of languages for the development of information systems at the conceptual level", In : IEEE Software, jan.1985, p.. 63-72, 1985.

[27] G. Simsion, G. Witt, Data Modeling Essentials, 3rd edition, Morgan Kaufmann, 2004.

[28] Standish Group , The Standish Group Report: Chaos, 1995.

[29] B. Shishkov, J. L. Dietz, "Deriving Use Cases From Business Processes, the Advantages of Demo", in Proceedings of ICEIS 2003, p. 138-146, 2003.

[30] M. Op 't Land, "Applying Architecture and Ontology to the Splitting and Allying of Enterprises", PhD Thesis, Technical University Delft (NL), 2008.

[31] Huysmans, P., "On the Feasibility of Normalized Enterprises: Applying Normalized Systems Theory to the High-Level Design of Enterprises", PhD Thesis, University of Antwerp, 2011.

[32] Van Nuffel, D., "Towards Designing Modular and Evolvable Business Processes", PhD Thesis, University of Antwerp, 2011.

[33] J. L. Dietz, "Enterprise Engineering Manifesto", 2010, Online available at: http://www.ciaonetwork.org/publications/EEManifesto.pdf.

[34] E. F. Codd, "A relational model of data for large shared data banks", Communications of the ACM, 13(6), 1970, p. 377-387.

[35] M. Lankhorst et al., Enterprise Architecture at Work - Modelling, Communication and Analysis, Springer-Verlag, 2005.

[36] T. Stahl, M. Volter. Model-Driven Software Development, Wiley, 2005.