Towards a System Requirements Specification Template that Minimizes Combinatorial Effects

Alberto Rodrigues da Silva[1], Jan Verelst[2], Herwig Mannaert[2], David Almeida Ferreira[1], Philip Huysmans[2]
{alberto.silva, david.ferreira}@inesc-id.pt, {jan.verelst, herwig.mannaert, philip.huysmans}@ua.ac.be

[1]IST/Universidade de Lisboa & INESC-ID
Lisbon, Portugal

[2]University of Antwerp & Normalized Systems Institute
Antwerp, Belgium

***Abstract*—This paper introduces the problem of combinatorial effects based on the evidence of many dependencies that explicitly or implicitly exist among the elements commonly used on system requirements specification (SRS). We start from the analysis and comparison of three popular SRS templates (namely IEEE 830-1998, RUP and Withall templates), mainly from the perspective of the constructs and models involved. Then we propose and discuss a set of practical recommendations to help defining a SRS template that may better prevent (to some extent) the referred problem.**

Keywords—System Requirements Specification (SRS); SRS templates; Combinatorial Effect (CE)

I. INTRODUCTION

A *combinatorial effect* (CE) is a kind of coupling and more specifically a ripple effect that can be defined independently of programming languages, systems development methodologies or frameworks used [1]. Furthermore, CE exhibits a highly harmful characteristic: it grows as the modular structure grows larger, which commonly occurs in practice over time.

In the requirements engineering (RE) community, "*system requirements specification*" or "*requirements specification*" (SRS) is a document that describes multiple technical concerns of a system, typically software systems but also blended software and hardware based systems [2,3]. A SRS is used throughout different stages of the project life-cycle to help sharing the system's vision among the main stakeholders, as well as to facilitate the communication and the overall project management and system development processes. A good SRS provides several benefits, namely [4,5,6,9,10]: establish the basis for agreement between customer and supplier; provide a basis for estimating budget and schedule; provide a baseline for validation and verification; and serve as a basis for future maintenance activities. A SRS is sometimes also integrated in legal documents surrounding project's Request for Proposals or Contracts. A SRS tends to follow a previously defined template that prescribes a given document structure (with chapters and sections) with supplementary practical guidelines. By definition, SRS templates should be adapted and customized to the needs of the organization involved. In any case, these templates prescribe the use of multiple (RE specific) constructs and models – corresponding to different views and perspectives of the system, for example goal-oriented, context, domain or use case models – that can be considered "modular artifacts", in the sense of their definition and reuse. However, there are several dependencies among and even intra these modular artifacts, which are important to minimize or prevent (to some extent).

In a recent work we discussed the result of our experiences in looking for CE at different levels in the RE-process [1]. The examples covered CE at the RE level based on the adoption of notations and techniques such as UML (classes and use case diagrams), DEMO/EO, and BPMN models. Those examples are relatively straightforward, but enough to show the omnipresence of such instabilities in the RE level. As a result, we described the need for a research agenda focusing on the systematic research into CE 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. Therefore, this paper has two other complementary goals. First, we compare the modular structures of three SRS templates in terms of the extent to which they prevent CE from happening. Second, we propose a set of practical recommendations to define a SRS template that would mitigate the referred problem.

Recently the RE community has been discussing the problem of modularization and composition of aspects and crosscutting concerns at the requirements level, designating them by CORE (concern-oriented requirement engineering) or AORE (aspect-oriented requirement engineering) [7,13,14]. Many contributions have been introduced by proposing techniques and models to support composition and integration of concerns in a way that prevents CEs. However, to the best of our knowledge, no further work discussing these issues at the SRS level has been proposed before. Studying CE at the level of SRS templates has the following advantages and implications: (1) The existence of CE at the RE-level would indicate that these modular artefacts exhibit limited evolvability and reusability. This is problematic in general because of the considerable cost associated with the requirement management process. It is sometimes referred that changing a requirements statement is many factors cheaper than applying the corresponding change to the resulting software. Nevertheless, CE could cause large SRS to also be very hard to develop and maintain, even prohibitively so. (2) Over time, the requirements of the system will change. However, updating the SRS is not as crucial to the business. Correspondingly, it will become increasingly likely that the SRS and the resulting system will become out of sync, resulting in business/IT-misalignment from a dynamic perspective. (3) RE can be considered the first step in the development of an information system, describing certain modular characteristics of the resulting system. If this initial

978-1-4799-6133-7/14 $31.00 © 2014 IEEE
DOI 10.1109/QUATIC.2014.22

description does not address modularity issues and dependencies, the software designer and developer will have to do so. However, with his/her detailed knowledge of the requirements, RE is best suited to judge aspects of modularity at the business- and requirements-level. The designer and programmer have no such detailed knowledge, which increases the chance of misalignments. (4) Other benefits related to minimizing these dependencies include a more elegant, coherent and easier way to design, specify and validate the document; less redundancy and inconsistency; and also the promotion of requirements reuse mechanisms.

This paper is organized in six sections. Section 2 introduces its background. Section 3 introduces and discusses the structure of three popular SRS templates. Section 4 proposes a SRS template based on a set of pragmatic recommendations. Section 5 discusses the impact of these recommendations and stresses their main benefits. Finally, Section 6 presents the conclusion.

II. BACKGROUND

A. Normalized Systems

Normalized Systems theory is aimed at studying how modular structures behave under change [11]. 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. The concept of stability demands that the amount of impacts caused by a change cannot be related to the size of the system and, therefore, remains 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 CE occurs.

Research has shown that it is very difficult to prevent CEs when designing software architectures. Namely, it has been proven that CEs are introduced each time one of four principles is violated, namely [11]: separation of concerns, data version transparency, action version transparency, and separation of states. The proofs of the principles show that the number of CEs will increase, making the software more complex and less maintainable, unless software is developed in a highly controlled way, ensuring that none of these principles are violated at any point of the development or maintenance phases, which is quite difficult to achieve in practice. Furthermore, it has been shown that software architectures can be built by constructing them as a set of instantiations of highly structured and loosely coupled design patterns, which provide the core functionality of information systems and are proven to be free of CE [12].

B. Constructs and Models used in SRS

SRS defines a set of statements that helps to share a common vision between business stakeholders and the development team, and facilitates the communication, negotiation and managing efforts among all the involved stakeholders. In general, requirements are specified in natural language due to their higher expressiveness and ease of use [10]. 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 often complemented by some sort of controlled or semi-formal language − usually graphical languages such as UML, SysML or KAOS −, which addresses different abstraction levels.

Additionally, requirements engineers often consider two distinct abstraction levels when organizing and specifying requirements: business and system levels. At the business level they define the purpose and general goals of the system, while at the system level they define with further detail the specific technical requirements of the system. The constructs considered at business level are the terminology, the goals that the system should satisfy, and the stakeholders who are the sources of these goals and requirements, but also business processes or business use cases. On the other hand, the main models considered in SRS at the system level are context model, domain model, functional requirements model and quality-attributes model. (A detailed analysis of these constructs and respective models is available in [1].)

III. SRS TEMPLATES

Considering the main constructs and models introduced in Section II, the following issue is how to organize all this information in a proper structure. It is particularly relevant to find out how to do that in a way that minimizes the CE regarding requirements change and reusability. Among a diversity of SRS templates found in the RE literature [4,5,6,8,9], we consider in this research the following templates: *IEEE Std 830-1998 (IEEE 830 template)* [9], *RUP Software Requirements Specification Template (RUP template)* [8], and *Withall template* [4]. Each template is structured in chapters and some complementary appendixes. It is relevant to discuss how the constructs and models are placed and managed in the context of these templates, which are still structured at the business and system levels as illustrated in Table 1.

At the business level, the main models are *terms, stakeholders and goals models*. In all the three templates these constructs are placed in their Chapter 1. We notice that there is not a specific section for explicitly defining the stakeholders model. Regarding the goals model, none of these templates recommend a particular textual or visual style to specify goals and their dependencies. However, they recommend that some kind of simple objectives of the system should be specified.

At the system level, the main models are context, domain, functional and quality attributes models. These models are placed into different sections. *Context models* provide the means to identify the boundaries of the system's scope and give a high-level overview of the system, particularly the context with other external systems and actors, or the context of the system with its subsystems and components. IEEE 830 and Withall templates recommend this vision (in section 2.1), without any particular notation.

Domain models allow capturing the key concepts and relationships underlying the system. These models allow a better understanding and specification of the problem domain, the concepts and respective terminology adopted throughout the project. In the analyzed templates it is not clear where the domain model is placed. Only the Withall template recommends its inclusion in section 2.4 (Main business entities) while in the IEEE 830 template it seems to be described in section 3.4 (Database Requirements), which is not exactly equivalent to domain models.

TABLE I. RELATING CONSTRUCTS AND MODELS WITH SRS TEMPLATES

			Templates		
Level	**Models**	**Constructs**	**IEEE 830**	**RUP**	**Withall**
Business	Glossary	Terms, definitions, acronyms	1.3 Definitions, acronyms, and abbreviations	1.3 Definitions, acronyms, and abbreviations	1.4 Glossary
	Stakeholders model	Stakeholders	-	-	-
	Goals model	Goals	1.2 Scope	1.2 Scope	1.1 Objective of the system
System	Context model	Systems, sub-systems, components, nodes	2.1 Product perspective	-	2.1 Scope
	Domain model	Entities, classes, objects	-	-	2.4 Main business entities
	Functional model	Actors, users	2.3 User characteristics	2.1 Use-case model survey	2.4 Main business entities
		Use-cases, user-stories	2.2 Product functions 3.2 Functional requirements	2.1 Use-case model 3.1 Use-Case reports	3. Functional Areas
	Quality attributes model	Qualities, metrics, utility values	3. Specific Requirements (all except 3.2)	3.2 Supplementary requirements	4. Non-Functional Requirements

Functional models allow specifying the functional and feature oriented requirements that the system should satisfy. There are several textual and graphical languages to support them. RUP template recommends the adoption of UML use case diagrams for organizing and visualizing these requirements, in Section 2.1 (Use-case model). Additionally, RUP advocates that each use case should be detailed in section 3.1 (Use-Case reports) by textual descriptions (e.g., using scenario descriptions) and other UML diagrams, such as activity or sequence diagrams. The IEEE 830 template includes a general specification of the functions or features required (in section 2.2 (Product functions) and the involved users (in section 2.3 (User characteristics)), but only in section 3.2 (Functional requirements) are the functional requirements described in detail. Conversely, the Withall template recommends the adoption of multiple chapters of functional areas (chapters 3 (Functional Areas) and beyond) depending on the size and complexity of the system and for a better way to organize and detail those functional requirements.

Quality attributes models allow specifying non-functional requirements related to the quality properties of the system, such as usability, performance, security, maintainability, and so forth. All these templates consider some chapter or section to include these requirements, namely: chapter 3 (Specific Requirements (all except section 3.2)) for IEEE 830 template; section 3.2 (Supplementary requirements) for RUP template; and chapter 4 (Non-Functional Requirements) for Withall template.

Finally, there are still other requirements (e.g., business or technical constraints), assumptions and, in some cases, exclusions that are also considered in SRS templates, usually in their initial or final chapters.

IV. PROPOSED SRS TEMPLATE AND RECOMMENDATIONS

In this section we propose a SRS template that prevents (to some extent) CEs to occur in the multiple dependencies that exist among these constructs and models. This template is relevant for a better organization of SRS documents as well as for their maintenance and evolvability. Table 2 shows its structure, which is designed according to a set of recommendations discussed in the following paragraphs (R1-R9). In this template the business level constructs are defined in Chapter 1 while the remaining chapters mainly describe the system level requirements. Chapter 1 includes the glossary, stakeholders and goals models. Having a section for explicitly identifying stakeholders is not proposed by the other templates; however, we consider it is relevant, in particular to better identify the sources and responsibilities for the goals and requirements.

Chapter 2 describes the general context of the system and its relations to other external systems or even can describe how the system is decomposed into subsystems. (We adopt "subsystem" as a general term for any concept in which we can decompose the original system; for example, a subsystem can be directly mapped to the concepts of components or modules according to some software architecture style or, into the concept of block as in SysML.)

The details of the system, eventually with its subsystems, are described in Chapters 3 and subsequent chapters. These chapters share the same structure based on five sections: Informational Entities, Actors, Functional Requirements, Use Cases, and Quality Attributes. Section 3.1 (Informational Entities) describes the entities corresponding to the main concepts and relationships underlying the system. Usually this description is complemented by diagrams, such as UML class diagrams, which help clarify the involved information. The name of those entities should be properly defined in the Glossary and be used consistently throughout the SRS document. Section 3.2 (Actors) defines the actors that interact directly with the system. Section 3.3 (Functional Requirements, FRs) includes one or more lists of FRs, which describe those functionalities or features that the system should satisfy. Section 3.4 (Use Cases) allows a detailed description of FRs and can be considered an optional section of the SRS. This section allows a clear identification of the actors that interact with the system through particular use cases, which can be even detailed through textual scenarios, user stories or other types of diagrams. Usually this description is complemented by diagrams, such as UML use case diagrams. Section 3.5 (NFRs at subsystem level) allows the definition of NFRs that are specific to the respective subsystem. The inclusion of this section is optional but can be important if the NFRs are specified with detail and directly assigned to one or a few subsystem's FRs.

Chapter N includes the definition of high-level NFRs, corresponding to the NFRs that are specified generically at the system or subsystem levels without much detail. Finally, and unlike other templates, the last chapter (i.e., Chapter N+1) allows the specification of composition and integration between the high-level constructs defined in previous chapters, namely between subsystems, between subsystems and NFRs, and between NFRs.

Below we discuss a set of recommendations that are underlying the proposed template.

R1. Separate business level from system level chapters. To avoid mixing both business and system levels constructs throughout the SRS document, a key recommendation is to separate them in different chapters. Together with R4 and R5, this recommendation allows a systematic assessment of whether a CE exists at business level or at system level.

R2. Split the system into subsystems, specify each subsystem into a self-contained chapter. For systems with a certain level of complexity or size, a good practice is to split the system into subsystems, which should be properly introduced and explained in Chapter 2 of the proposed SRS template and visually described by a context or block diagram. Each subsystem should be described in an independent chapter, self-containing the following constructs: informational entities, actors, FRs, use cases, and even NFRs (defined at subsystem level, if relevant). There are some advantages to splitting the system into subsystems and organizing the main chapters around this notion of subsystem. First, splitting the system into subsystems is a natural and common approach to deal with complexity. Second, defining a chapter corresponding to each subsystem is also a natural and elegant way to organize and maintain a SRS document. This structure promotes a better focus on the knowledge and functional aspects of a given subsystem by defining its boundary and its external dependencies and interfaces. Third, this structure also facilitates the distribution and assignment work through different teams or individuals: different teams can work independently on different subsystems specifications. Fourth, this structure prevents the existence of CE at the requirements specification level. For example, if we need to add, change or delete a given construct (e.g., entities, FRs or NFRs) the impact of that change tends to be managed only at that chapter and not spread outside of it.

R3. NFRs are specified in an independent chapter (but can also be specified at a subsystem chapter, if they are detailed enough) and can be derived from Goals. NFRs such as security, performance, usability, or maintenance, are complex properties that the system as a whole or a subsystem should satisfy. Because several of these properties cannot be specified and checked for just a part of the system, they should usually be specified in an independent chapter that would not be directly attached to any specific subsystem. However, there are also situations when a NFR should be directly assigned to just one (or a restricted number of) FR, typically in those situations when requirements are defined in more rigorous and detailed ways. In this situation, these NFRs should be defined in the corresponding subsystem chapter. Additionally, it is a good practice to explicitly define which goal each NFR derives from. The links between goals and NFRs are not mandatory but help to keep the history and the rational of these requirements.

R4. Decouple Actors from Stakeholders. Stakeholders are defined at the business level and correspond to individuals or organizations such as customer, sponsor, marketing director, k-users, project manager, performing organization, or development team. Stakeholders have a positive or negative interest in the project and ultimately will be affected by its results. Actors interact with the system being either end-users or external systems. Still, actors are relevant when detailing FR through the descriptions of use cases. Stakeholders and actors are constructs defined at different abstraction levels and they do not should be blended or confused. However, some actors can be derived from specific stakeholders (e.g., in a ERP system, the actor "Financial Manager" can be derived from the stakeholder "Financial Manager").

R5. Decouple Functional Requirements from Use Cases. FR specifications can follow different writing styles, being more verbose or brief, but some aspects should be guaranteed, such as: they should be univocally identified, promoting a better identification and also traceability among FRs; they can be derived from functional goals; their textual specifications should use consistently the terms defined in the Glossary as well as the entities defined. On the other hand, use cases are a complementary technique that allows organizing and detailing those requirements but, depending on the development process, they can be optional or mandatory. For example, RUP or Extreme Programming strongly advocates the adoption of use cases or user stories for that purpose. However, in some situations, particularly at the early stages of the project and for the sake of simplicity or economy, use cases should not be specified at all.

R6. Define a Glossary of Terms, splitting application domain Terms from project or system specific Terms. Section 1.2 (Glossary) can be split into two subsections: one with domain specific terms and another one with a project or system specific terms. That way, the first subsection can involve terms reused throughout different projects that involve the same application domain.

R7. Specify the integration of concerns. The three SRS templates mention a number of concerns (mainly dividing functional from non-functional concerns), but have limited or no attention for the integration and composition of these and other crosscutting concerns. However, it is precisely at the level of this integration of different concerns that many CE exist, as the previous section illustrates. The Chapter N+1 allows the specification of the integration between the high-level constructs defined in previous chapters, namely between subsystems, between subsystems and NFRs, and between NFRs themselves. Some CORE or AORE approaches may be adopted to better support this specification [7].

TABLE II. PROPOSED SRS TEMPLATE

Level	Template Structure	Comments
Business	1. Introduction	*Chapter 1 presents the Business View*
	1.1 Purpose of the document	
	1.2 Glossary	***Glossary model**, with lists of domain-specific and system-specific terms*
	1.3 Stakeholders	***Stakeholders model**, with generic stakeholders and concrete stakeholders*
	1.4 Goals	***Goals model**, e.g., using visual notations such as i* or KAOS*
	1.7 References	
	1.8 Organization of the Document	
System	2. System overview	*Chapter 2 presents the high-level System View*
	2.1 Context	***Context model** defines how the system is organized in subsystems and how it interoperates with other external systems*
	2.3 Key Assumptions	
	2.4 Technical Constraints	*e.g., contraints related the use of some tool, database server, or development process*
	2.4 Main Exclusions	
	3. Subsystem A	*Chapters 3, 4, 5, .. present the Subsystem Views*
	3.1 Informational Entities	***Domain model**, e.g., using visual notations such as UML class diagrams or ER*
	3.2 Actors	*List of actors*
	3.2 Functional Requirements	*List of functional requirements*
	3.3 Use cases	***Use case model**, e.g., using visual notations such as UML use cases and textual narratives*
	3.4 NFRs (at subsystem level)	***NFR Models**, using e.g. QA Scenarios for describing subsystem-specific NFRs*
	…	
	N. High-Level Non-Functional Requirements	*Chapter N presents **NFR Models**, using e.g. QA Scenarios, for describing NFRs defined generically at (sub)system levels*
	N.1 NFR 1	
	N.1 NFR 2	
	…	
	N+1. Concerns Integration	*Chapter N+1 presents the integration and composition of concerns, based on some CORE or AORE approach*
-	Appendixes	*Complementary data to the SRS, for example: Versions of the Document; Traceability Matrixes*

R8. Define domain-specific anticipated changes. The previous guidelines address domain-independent concerns. However, it was previously noted that the three SRS templates do not contain any domain-specific concerns. This means that the requirements engineer has to identify the concerns on a project-by-project basis. Once this is done, or helped by, a systematic study of the identification of high-frequency anticipated changes in the domain. Typically, domain-specific concerns are separated in such a way that these anticipated changes can be contained. In order to facilitate this, SRS templates should stipulate that the requirements engineer should systematically study these anticipated changes.

R9. Document the appropriate level of abstraction. All three templates mention NFRs, such as performance, security, or usability. A relevant challenge is getting the abstraction level of such requirements right. On the one hand, abstraction levels are sometimes very high in examples such as: "The response time of the system to user interactions should be less than 3 seconds". From a technical perspective this requirement is very inflexible. Usually, a few interactions are far more computationally intensive than others, and stating a response time at a very high level of granularity (across all interactions), forces the designer to over-engineer the system at a considerable expense. From this perspective, it would be preferable that response times are set at a lower level of granularity, e.g. per type of interaction, or perhaps even per (group of) screen(s). On the other hand, sometimes the abstraction level is very low, stating that 'data caching should be used to limit the response time of the system to customer inquiries'. Again, this requirement forces the designer to take technical decisions which may be technically suboptimal. SRS templates should specify at what abstraction level the NFRs should be written, and this guideline is directly related with R3.

V. DISCUSSION

We have to state that the above recommendations are the result of *observation* and *reflect our experience* in the field both in practice and research. On one hand, we have many 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 [11,12]), requirements specification languages (e.g., the ProjectIT-RSL [15] or RSLingo approaches [16]), and the alignment between RE and MDE fields [17]). The scope and contribution of this template reflects this blended experience and in particular the recent design of the RSL-IL language that incorporates many of these recommendations [18]. Additionally, some of these recommendations are also implicitly seen in the analysed templates. For example, R1 and R2 are typically addressed in the IEEE 830 template by hierarchies of requirements documents; R3 is also supported by Withall template; R5 and R6 are common guidelines for requirements documents.

The proposed template is not a closed template and should be adapted and customized to the needs of the involved organizations on their specific needs. Namely, some adaptations can occur on the top of the proposed template without a significant impact, for example: (1) move the glossary of terms and the references sections from Chapter 1 to the Appendixes in the end of the document; or (2) for simple software systems or for SRS defined at an early stage of a project, you can merge the chapters focused on the functional concerns of the different subsystems into just one (i.e., merge Chapters 3, 4, … N-1 into only one chapter); or still in this situation, (3) you can avoid describing detailed information of the subsystem's functional description (e.g., avoiding to describe use cases and scenarios) because it can be just enough to enumerate the main FRs for each subsystem.

This template helps to minimize and prevent CE but it does not claim to avoid them. On the contrary, there are still multiple dependencies among the constructs and models used in SRS. Figure 1 suggests the main perceived dependencies. Additionally, there are still other dependencies that are implicitly illustrated in the figure for the sake of readability. For example, there are dependencies from several constructs (e.g., stakeholders, actors, informational entities) to the terms defined in the glossary, meaning that the name of these constructs can or should be properly defined in the SRS glossary. Another example of implicit relationships are the intra-dependencies that still exist between FRs, NFRs, or FRs and NFRs.

Overall, in comparison with others, this template may promote a better structure concerning the ease of change and the requirements' evolvability, and perhaps more important than the template itself are the discussed recommendations.

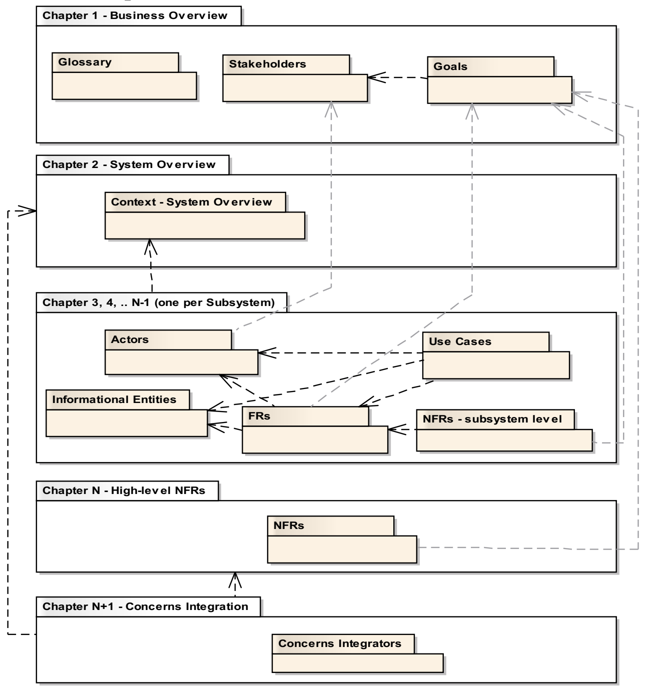
Chapter 1 - Business Overview
Glossary
Stakeholders
Goals
Chapter 2 - System Overview
Context - System Overview
Chapter 3, 4, .. N-1 (one per Subsystem)
Actors
Use Cases
Informational Entities
FRs
NFRs - subsystem level
Chapter N - High-level NFRs
NFRs
Chapter N+1 - Concerns Integration
Concerns Integrators

Fig. 1. Main dependencies inside the proposed SRS template.

VI. CONCLUSION

Normalized Systems theory answers how modular structures behave under change. Originally 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. In this paper, we started from this background and our experience in defining and using Requirements Specification Languages, and discussed how the *structure of SRS* can influence the requirements management process, in particular, the change and evolvability of SRS documents. To support this research we surveyed the key constructs and models commonly used in requirements specifications, and we analyzed three popular SRS templates and discussed the matching of these constructs and models in those templates. Based on some examples, we observed that there are many dependencies among those constructs and that, in some cases, they are scattered throughout the SRS, which means severe limitations in what evolvability is concerned. Finally, as a result of this research, we proposed a SRS template, with a companion set of recommendations, that helps reduce CEs at RE level. Without such control of CE, SRS will be insufficiently evolvable to deal with the demands for change that organizations are increasingly facing.

ACKNOWLEDGEMENTS

This work was partially supported by Portuguese funds through FCT – Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2013 and DataStorm Research Line of Excellency funding (EXCL/EEI-ESS/0257/2012).

REFERENCES

[1] Verelst, J., Silva, A.R., Mannaert, H., Ferreira, D.A., Huysmans, P.: Identifying combinatorial effects in requirements engineering. in Proceedings of EEWC 2013, LNBIP, Springer, 2013.

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

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

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

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

[6] Robertson, S. and Robertson, J.: Mastering the Requirements Process, 2nd edition. Addison-Wesley, 2006.

[7] Gray, J., Bapty, T., Neema, S. and Tuck, J.: "Handling Crosscutting Constraints in Domain-Specific Modeling," Communications of the ACM, 44(10), pp. 87-93, 2001.

[8] IBM: Rational Method Composer and RUP on IBM Rational developerWorks, 2013. http://www.ibm.com/developerworks/rational/products/rup/

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

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

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

[12] Mannaert, H., Verelst, J. and Ven, K.: "Towards evolvable software architectures based on systems theoretic stability," Software Practice and Experience (42:1), pp. 89-116. Wiley, 2012.

[13] Moreira, A., Araújo, J., and Rashid, A.: "A concern-oriented requirements engineering model," in Proc. CAiSE'2005, Springer, 2005.

[14] Tun, T. T., et al: "Specifying features of an evolving software system," Software: Practice and Experience 39(11), 2009.

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

[16] Ferreira, D., and Silva, A. R.: "RSLingo: An Information Extraction Approach toward Formal Requirements Specifications", in Proc. of the MoDRE'2012. IEEE Computer Society, 2012.

[17] Silva, A. R. et al: "Integration of RE and MDE Paradigms: The ProjectIT Approach and Tools", IET Software Journal, 1(6), IET, 2007.

[18] Ferreira, D., and Silva, A. R.: "RSL-IL: An Interlingua for Formally Documenting Requirements", in Proc. of the MoDRE'2013. IEEE CS, 2013.