Requirements Evolution Understanding Formally Software Engineering Processes within Industrial Contexts

Consiglio Nazionale delle Ricerche
Bando n. 203.15.11

Massimo Felici

LFCS, Division of Informatics, The University of Edinburgh
JCMB, The Kings Buildings, Mayfield Road
Edinburgh EH9 3JZ, United Kingdom
tel. +44-131-6505899
fax. +44-131-6677209
mas@dcs.ed.ac.uk

February 12, 2002

Abstract

This report considers software requirements evolution within industrial production environments. Any production environment consists of diverse stakeholders interacting through processes focused on particular aspects of software production. Thus, software is the result of cooperating processes in which the human is the main actor. Effective cooperation requires stakeholders to understand their different viewpoints on the processes. Stakeholder interaction, cooperation and negotiation result in shifts in the grounds for agreement. These shifts drive requirements evolution. This report takes requirements evolution as an unavoidable feature of software production. Classically, requirements evolution is seen as an error in the engineering process. By contrast, this report considers requirements evolution as an essential feature of good design processes.

This report and the work it describes were funded by a grant of the Italian National Research Council (CNR), Bando n. 203.15.11, Research Program: Requirements Evolution: Understanding Formally Engineering Processes within Industrial Contexts.

Acknowledgements

I thank Stuart Anderson for his supervision on my work. I thank, moreover, the industrial partners, who provided the case studies. Due to the confidentiality agreement with the industrial partners I can not provide any further detailed information. Despite this the work still remains valuable and the results are clearly expressed. This work has been conducted within the DIRC project[1]. The DIRC project has been funded by the UK EPSRC (Engineering and Physical Sciences Research Council). Finally I thank all the DIRC people and colleagues who collaborated and provided valuable input in related discussions, in particular, I thank John Dobson, Juliana Küster Filipe and Robin Williams.

This report and the work it describes were funded by a grant of the Italian National Research Council (CNR), Bando n. 203.15.11, Research Program: Requirements Evolution: Understanding Formally Engineering Processes within Industrial Contexts.

Related Publications

This work has been partially published in the following papers.

[4] Stuart Anderson, Massimo Felici, "Requirements Evolution: From Process to Product Oriented Management". In Proceedings of the 3rd International Conference on Product Focused Software Profess Improvement, Profes 2001, Kaiserslautern, Germany, September 10-13, 2001, LNCS 2188, ©Springer-Verlag 2001, pp. 27-41.

[23] Massimo Felici, Juliana Küster Filipe, "Limits in Modelling Evolving Computer-based Systems". To appear in Proceedings of the ACM Symposium on Applied Computing, SAC 2002, March 10-14 2002, Madrid, Spain.

[1] Stuart Anderson, Massimo Felici, "Controlling Requirements Evolution: An Avionics Case Study". In F. Koornneef, M. van der Meulen (Eds.), Computer Safety, Reliability and Security, 19th International Conference, SAFECOMP 2000, Rotterdam, The Netherlands, October 2000, Proceedings, LNCS 1943, ©Springer-Verlag 2000, pp. 361-370.

[1] http://www.dirc.org.uk/

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This report considers software requirements evolution within industrial production environments. Any production environment consists of diverse stakeholders interacting through processes focused on particular aspects of software production (e.g., management, designing, testing, etc.). Thus, software is the result of cooperating processes in which the human is the main actor [42, 89]. Effective cooperation requires stakeholders to understand their different viewpoints on the processes [44, 52, 64, 86, 88, 92]. Stakeholder interaction, cooperation and negotiation result in shifts in the grounds for agreement. These shifts drive requirements evolution. This work takes requirements evolution as an unavoidable feature of software production. Classically, requirements evolution is seen as an error in the engineering process. By contrast, this report considers requirements evolution as an essential feature of good design processes.

Stakeholders interact as participants in the processes of a software development environment. These interactions result in changes in stakeholder knowledge. These changes are captured and disseminated over the project as requirements evolution [33, 68]. Requirements evolution triggers a sequence of events, which allows changes to propagate throughout the development process. In any development process the definition of requirements is the first phase and it is always crucial for the success of the project (e.g., in terms of cost, timing, customer satisfaction, etc.). As a software project progresses changing requirements becomes increasingly expensive and project risk increases [8, 9, 11]. Each requirements change affects the success of the project as well as system characteristics (e.g., dependability, safety, reliability, usability, etc.) that are crucial for the system functionalities (e.g., Air Traffic Control, Electronic Engine Control, Medical Systems, Smart Card Systems, etc.). These issues motivate an increasing interest in requirements engineering [7, 19, 84]. The deeper the understanding of stakeholder interaction, the better the control of requirements evolution. The problem is how to match business processes, development processes and product features in order to enhance our ability in understanding and controlling requirements evolution by monitoring stakeholder interaction through requirements.

This work aims to enhance our ability to understand and control the evolving nature of requirements. In contrast to the process-centred approach taken in current requirements engineering practice [87, 76, 95], we take a product-centred approach. Process issues are captured in the product as it is developed. Our approach originates in the empirical investigation of industrial case studies of evolving products and their requirements [1, 2, 25, 26]. In these studies we aim to develop a detailed account of the cooperative processes adopted by stakeholders. The underlying hypothesis of this report is that stakeholder interaction in cooperative processes is a powerful driver of requirements evolution.

1.1 Related Work

Current practice in requirements engineering [76, 81, 87, 95] identifies a set of methodologies, tools, processes that aim to specify software (system) requirements. Two big research and practice areas can be identified within requirements engineering, namely, specification and verification[1]. The former aims to improve our ability in specifying requirements. The latter aims to improve our ability in verifying requirements characteristics (e.g., correctness, completeness, etc.). Everything concerning requirements evolution is usually covered under another area, namely, requirements management. The name itself emphasises process oriented approaches. There has been little attention from the product point of view to devise approaches supporting Requirements Evolution. Most of the work in requirements management focuses on process aspects. Recent research [32, 33, 68, 90] in requirements engineering points out the evolving nature of requirements. The PROTEUS project [33, 68] takes into account requirements evolution from a business viewpoint identifying the origins of changing requirements into stakeholders and business environment. Any specific environment producing software is characterised by changing requirements, therefore any attempt of stopping changes [32], hence requirements evolution, will fail triggering process and software degradations. The understanding of requirements evolution may be improved by empirical investigations [1, 2, 32, 90] of industrial case studies, which analyse how requirements evolve in practice. Evolution should be considered differently than reuse. Requirements reuse should take into account a trade off between process and product viewpoint [48, 49]. Even the solution of reusing requirements has to be carefully adapted to specific software contexts [48, 49].

The COHERENCE [89] and REAIMS [88, 92] projects encourage the application of hybrid approaches early in the development process even before the specification of requirements. In particular, the integration of different viewpoints seems an effective way of merging different perspectives related to different stakeholders and engineering processes [86, 88, 89]. It comes out that even with hybrid approaches focusing on the process, requirements evolution still remains one of the critical point throughout the software life cycle.

[1] Other research areas in requirements engineering are elicitation and analysis, but here we are not referring to these areas.

The evolving behaviour is not bounded at the requirements level. The FEAST project [53] has empirically identified a set of laws for software evolution. But the relation between requirements evolution and software evolution [5, 53, 43] needs further clarification. Further investigations of this relationship will improve our ability to assess the impact of requirements changes. Moreover, the widely held view that traceability [20, 21, 41, 69] resolves requirements evolution turns out to be insufficient. Requirements traceability [41] has been recognised to be an important aspect to deal with evolution. Traceability represents not only an additional information to be collected (and maintained), but also a strategic policy involving an entire software organisation [69]. Traceability maintenance relies on strict processes and tools, which allow to identify relationships between the software organisation and the software product [41]. The combination of traceability with other relationships is needed to enhance our ability in specifying and maintaining requirements. Requirements and Software Evolution has also been considered from a theoretical point of view in [61, 98], but there has been little attention to integrate theoretical work with product-line features.

Many recent research projects [17, 18, 62, 72, 73, 74] have progressed the state of the art in Requirements Engineering and the understanding of evolutionary behaviours in the software life cycle. But the current practice in industry requires additional research actions. Past projects point out the need to shift to hybrid process-product oriented methodologies, which shall be defined according the needs identified according to massive empirical investigations of real case studies.

1.2 Report Outline

This report is structured as follows. Chapter 2 provides a detailed account of evolution of computer-based systems. A taxonomy emphasises the different aspects of evolution. Three different case studies are presented in the following chapters. Chapter 3 shows an empirical investigation of an avionics industrial safety-critical case study. The empirical analysis provides input to model evolutionary aspects of requirements. Chapter 4 shows an analysis of requirements viewpoints in an industrial case study. The different viewpoints provide different way of taking into account requirements evolution. Divergencies of requirements viewpoints may identify issues into requirements and into the processes dealing with their evolution. Chapter 5 shows a case study pointing out limits in modelling computer-based systems. Finally chapter 6 draws the conclusions of this work.

Chapter 2

A Taxonomy of Evolution

Computer-based systems represent continuously evolving citizens of the modern electronic mediated society. They are continuously (re-)designed and (re-)deployed in order to capture environmental evolution. The necessity to capture environmental evolution has been recognised by design life cycles giving rise to evolutionary developments [5, 10, 67, 85]. Thus evolution turns to be an inevitable and necessary phenomenon of computer-based systems. Unfortunately experience shows that evolution affects computer-based systems such that triggering degradation raising catastrophic failures [56, 66, 91]. Hence evolution affects system dependability.

Laprie defines *Dependability* as "that property of a computer system such that reliance can justifiably be placed on the service it delivers". The attributes (i.e., Availability, Reliability, Safety, Confidentiality, Integrity and Maintainability) refining dependability differently emphasise aspects of dependability. The dependability tree moreover identify the means (i.e., Fault Prevention, Fault Tolerance, Fault Removal and Fault Forecasting) to achieve dependability in a computer systems by interfering with the impairment (i.e, faults, errors and failures) mechanisms [50, 51]. Figure 2.1 shows the fundamental chain[1] (or cycle) of faults, errors and failures representing the basic mechanism increasing the risk of undependability of computer-based systems. The dependability means tackle this cycle in different ways according to the point in which they attempt to break the cycle. Both the basic impairment and the dependability means imply *evolution* of computer-based systems. This interpretation draws evolution as an orthogonal concept to dependability. Thus evolution becomes another means to achieve dependability of computer-based systems. Hence dependability is an emergent system property eventually achieved by evolution. On the other hand evolution is also a major source of risk of undependability of computer-based systems.

Evolution is a broadly accepted concept, but people actually refer to different evolutionary aspects. This paper contributes to clarify the different viewpoints

[1] Fenton and Pflegger [27] propose a similar model based on the sequence errors, faults and failures corresponding to the Laprie's sequence of faults, errors and failures.

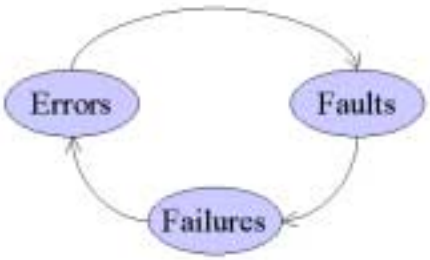

Figure 2.1: The cycle of undependability impairment.

about evolution. In practical terms a taxonomy may help to improve communication and reduce misunderstandings among people interested about evolutionary aspect of computer-based systems. A taxonomy may furthermore help to identify inconsistencies due to the different understandings about evolution. This chapter is structured as follows. Section 1 identifies a layered structure to analyse evolution. Each layer refers to a different type of evolution. These layers point out that each evolution can differently contribute to (un)dependability. Section 2 then reviews different evolutionary models pointing out our ability in representing evolutionary aspects of computer based systems. We then discuss the relationship between the taxonomy of evolution and dependability in Section 3.

2.1 Evolutionary Layering

This section analyses different evolutions occurring in computer-based systems. These evolutions identify different layers in which evolutionary phenomena occur. As evolutionary layers range from software evolution to organization evolution, our analysis ranges from *hard* to *soft* evolution. Figure 2.2 shows the different layers representing an ideal structure to analyse computer-based systems with respect to evolution. The lowest level consists of *Software Evolution* and analyses evolution from a software product viewpoint. The second level, namely *Architecture (Design) Evolution* takes into account how evolution is perceived at the design level. The third level, namely *Requirements Evolution*, takes

into account evolution for the requirements of computer-based systems. This is a natural intermediate viewpoint, because requirements are used as a means of interaction among stakeholders. Requirements are used during negotiation between the customer and the system provider. Nevertheless requirements represent the major information for designers and software engineers. Hence the requirements level is a natural gate where to capture information about evolution of computer-based systems. The remaining three levels, namely, *System Evolution*, *Human(-Machine) Evolution* and *Organization Evolution*, take into account a systemic viewpoint of computer-based systems. These three levels emphasises human factor with respect of evolution. The three-level separation allows to investigate human related evolution, but also evolution as a whole for computer-based systems.

Figure 2.2: Evolutionary layers.

Software Evolution. The problem of software evolution has been extensively investigated by Lehman et al [54, 55]. The investigation of software evolution as a natural phenomenon has lead to the identification of *E-type programs* (software systems). According to the Lehman's laws of software evolution, E-type programs continuously evolve in oder to be satisfactory (for users) and (need) to accommodate environmental changes. E-type programs have an increasing complexity if maintenance is not performed and represent multi-level, multi-loop, multi-agent feedback systems. A main challenge in engineering software still remains to be the management of evolution [53]. So far software evolution (and in general evolution) has been considered a management process [93] with little emphasis on product evolutionary features. Research in specific context, i.e.,

Object-Oriented, has identified patterns of software evolution [29]. This experience shows that evolution is not just a process feature, but it is also a product characteristic. Nevertheless evolution can be considered an inherent product characteristic that may enhance quality as well as dependability in general.

Architecture Evolution. At the Design level, it is still vaguely understood what is the role of the system architecture with respect to evolution. Experience shows that Architecture is risky and expensive to change, even if the process and the architecture itself are well understood [47, 83]. According to our experience architecture usually represents the stable part of those systems for which there are stringent safety and security requirements [1]. This is consistent with related research in requirements engineering [68] emphasising that stable requirements have origin in the core of the business. Hence the architecture stability depends to some extent to its relation with the core of the business. The more constrained is the business, the more evident is the relation between architecture stability and business core.

The architecture represents a central concept for product-lines [12, 94]. Our ability in predicting evolution is crucial for the definition of product-lines, in fact, product-lines implement the extent to which a system can be adapted to future needs. Thus a product-line architecture represents a trade-off between generality and specificity. In practice, architectures and product-lines evolve around variation points identified in the design of the product-lines. However it is often misunderstood the definition of architecture evolution. There exist in general two (three) types of evolution: the architecture evolves or architecture's components evolve (everything evolves).

Requirements Evolution. Practice shows that requirements evolve [1, 2, 4, 35, 36, 68, 90] affecting system dependability and process effectiveness. It is impossible to frozen requirements, but we should (and could) analyse the extent to which requirements evolve in order to identify the stable ones and the most likely to change [1]. Requirements evolution has been mainly considered a management problem, rather than a necessary feature of computer-based systems and their life cycles. Requirements Engineering literature has little emphasis on product features [36, 76, 87, 95]. Our empirical analysis of an industrial case study points out that there exist product features enhancing our understanding of (requirements) evolution. The empirical analysis of industrial case studies aims to identify structures in Requirements Evolution. We can distinguish different types of structures characterising Requirements Evolution. For instance, dependencies over requirements [4] identify relationships, which may be redefined across subsequent releases. This may allow to minimise dependencies and to refine impact analysis of changes into requirements. Other structures are those defining the type of changes into requirements, e.g., adding, deleting and modify requirements change the requirements documentation depending on the order these operations are performed, the number of changes and the template used for specifying requirements (e.g., [39, 75, 76]). These structures allow to

define the evolution process itself in a structured way. Thus an instance of requirements evolution may be characterised in terms of requirements dependencies and sequence of requirements changes. The Proteus Project [68] classifies the origins of stable and changing requirements and proposes a conceptual framework for changing requirements. This framework aims to support three main strategies to deal with requirements changes. The proposed strategies are *reducing changes*, *facilitating incorporation* of changes and *identifying needs* for change as early as possible. These strategies make use of enabling technologies like predictive analysis of changes, traceability of requirements, formal framework for representing and reasoning about changes, and prototyping. Guidelines and checklists support the strategies dealing with requirements changes.

System Evolution. Software is just a part of computer-based systems. Our view then focuses on evolution of computer-based systems as a whole. This means that our analysis takes a *holistic* viewpoint with respect to evolution. Figure 2.3 shows the SHEL model [22], which supports a systemic view defining any productive process as performed by a combination of *Hardware* (e.g., any material tool used in the process execution), *Software* (e.g., procedures, rules, practices, etc.) and *Liveware* (e.g., end-users, managers, etc.) resources embedded in a given *Environment* (e.g., socio-cultural, political, etc.). The knowledge required to perform a specific process can be considered as being distributed among the system resources. Thus, a productive process may be regarded as an instantiation of the SHEL model for a specific process execution. The systemic view of the SHEL model takes into account not just the system in terms of hardware and software, but also those aspects (e.g., procedures, practices, human roles, interaction, help in breakdown-situations, etc.) related to Liveware resources. Therefore, SHEL systems represent a trade-off among Hardware, Software and Liveware resources in a given Environment. A systemic view, like that one of the SHEL model, emphasises the need to take into account not just evolution into parts, but also evolution in the whole. Hence, software evolution, e.g., should trigger evolutions into all the other parts of the system. An effective management of this evolutionary chain allows to keep system artefacts unaffected. On the other hand evolution across resources may allow new artefacts to emerge as resulting behaviour of the evolved systems. The remaining paragraphs analyse evolution for Liveware resources.

Human(-Computer) Evolution. A systemic view of computer-based systems emphasises the presence of human. Hence computer-based systems are form of socio-technical systems. The evolution of computer-based systems depends on human factor as well as technical aspects. The variety of human behaviours can not be captured by any modelling, but recent promising research has identified some mechanisms explaining how human perceive machines in order to acquire computational artefacts and accomplish specific tasks. *Social Learning* [96] consists of two main processes named *Innofusion* and *Domestication*. Innofusion is a practical activity of learning by trying [28] that allows

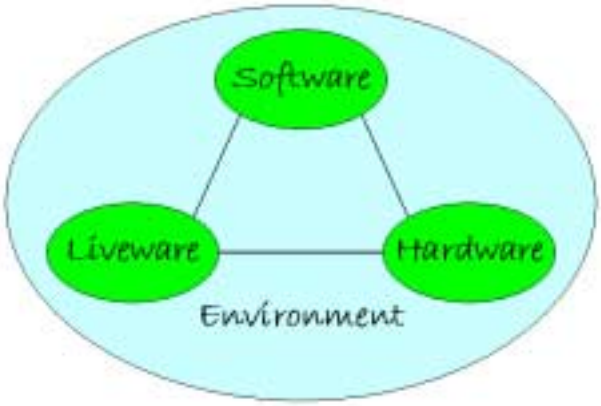

Figure 2.3: The SHEL model.

to customise the computational artefacts (behind a computer-based systems) to meet social needs. The underlying hypothesis is that workers, individually as well as collectively, develop more efficient ways of employing machinery through their experience from usage. This kind of learning curve effect is well-known. Though often taken to be limited to the initial introduction of new equipment, it is now recognised that learning by doing may improve the efficiency of production over a very long period of time. This discussion of the user role also points out one sense of configuring as the creative (indeed perhaps playful) rearrangement of a number of elements. Whilst Innofusion usefully captures the contribution of these distributed learning processes to the evolution of the artefact, Domestication addresses the creative role of the user in integrating new artefacts within their everyday activities and meanings. Domestication, where people learn by situated activity, is a practical activity of learning by interacting. This activity stresses the complex interaction between technology supply and use by applying evolutionary metaphors of the generation of variation and selection of artefacts.

Social Learning points out that socio-technical systems are characterised by emergent behaviours. That is, in communities of practices emergent behaviours give rise to groups and social contexts where each individual build his/her own identity within that context, by creating a process of differentiation in himself and in the environment that are instrumental to the production of emergent behaviour. At the core of this differentiation process there is the creative process of Artefact Design. The complex settings of socio-technical systems have been recognised by the recent approach of *Distributed Cognition* [63], which analyses

how humans work, operate and create external and internal artefacts. This approach re-elaborates the long lasting thesis that human cognition is mediated by artefacts (rules, tools, representations), which are both internal and external to the mind. The central tenet of the Distributed Cognition approach is that knowledge is distributed across people and artefacts; cognition is not a property of individuals but rather a property of a system of individuals and artefacts carrying out some activity. According to these theoretical assumptions, human activity and artefacts are the two inseparable sides of the same phenomenon: human cognition.

Organization Evolution. The above paragraphs point out the strong connection between technical systems and social aspects, hence socio-technical systems. The evolution of socio-technical systems influences the organizational context as well. Thus system evolution implies an organization co-evolution, and vice versa. The mechanisms driving organization evolution characterise the new electronic mediated society [13, 16, 57]. These mechanisms are similar to that ones driving the evolution of socio-technical systems. The link between socio-technical evolution and organization evolution has not been yet fully registered and still remains challenging for future research. The study of organizational evolution involves the understanding of socio-technical evolution as well as business understanding.

2.2 Evolutionary Modelling

This section reviews models capturing evolution at each layer of the taxonomy. At software level, E-type programs provide a general model to analyse software evolution [54, 55]. Software Metrics capture to some extent the evolutionary properties of software relating them to its quality as well [27, 37, 38, 40, 80]. Metrics quantify how software has been modified, e.g., in terms of Lines Of Code (LOC). Specific metrics assess software reliability and probabilist models, namely Reliability Growth Models, predict the evolution of software reliability [60]. With respect to evolution Software Reliability Growth Models present some applicability limits [24, 59]. This is mainly due to their assumptions, which, in some cases, do not fit adequately the real conditions of the software development process. The validity of the operational profile can affect the effectiveness of these models. The operational profile can be affected by several evolutionary factors. For instance, software can have different users, each of which being characterised by a different operational profile. That is, users differently learn and customise a computer-based system according to their needs. Even the software architecture can change during development and testing, or can be modified in operation, leading to substantial changes in the operational profile. All these factors make the operational profile unknown or very difficult to estimate.

Patterns of software evolution have been identified mainly for Object Oriented contexts [29]. Three main patterns have been identified, namely, *Software*

Tectonics, *Flexible Foundations* and *Metamorphosis*. Software Tectonics shows how continuous evolution can prevent cataclysmic upheaval. Flexible Foundations catalogues the need to construct systems out of stuff that can evolve along with them. Metamorphosis shows how equipping systems with mechanisms that allow them to dynamically manipulate their environments can help them better integrate into these environments [29]. The recognition of patterns for software evolution is in line with the architecture evolution. Architecture evolution is risky [47], if it is not controlled around identified variation points often characterising product lines [12]. We are currently investigating how to enhance our ability in modelling evolving computer-based systems [23]. Our analysis points out some limits in modelling approaches. Tackling these limits may furthermore enhance our ability in integrating different requirements (viewpoints) [26].

Logical models for requirements evolution have been proposed [68, 30, 98]. The Proteus project [68] proposes a *goal-structures framework* based on the fundamental components of goals, affects, facts and conditions. The proteus' framework allows to specify scenarios as set of conditions (believed to be valid) identifying operational profiles for a model of the system in terms of facts (i.e., statements taken to be true). The set of conditions and the systems' facts assess whether desired systems properties, expressed in terms of goals or effects, arise. Thus the framework provide a logical formalism for cause-effect relationships to specify system requirements. The framework has an expressiveness of first-order logic and allows resolution in prolog-style. Another formal framework [97, 98] models requirements by a non-monotonic logic and requirements evolution consists of mapping one such model to another. Hence requirements evolution consists of a sequence of non-monotonic logics. The framework is then extended to capture the negotiation between functional and non-functional requirements [30]. These frameworks show that logical formalisms may help to analyse requirements evolution. The next step is to integrate logical models with empirical information in order to capture context's features.

The widely held view that traceability [20, 21, 41, 69] resolves requirements evolution turns out to be insufficient. Requirements traceability has been recognised to be an important aspect to deal with evolution [6, 41]. Traceability represents not only an additional information to be collected (and maintained), but also a strategic policy involving an entire software organisation [69]. Traceability maintenance relies on strict processes and tools, which allow to identify relationships between the software organisation and the software product [41]. The combination of traceability with other relationships is needed to enhance our ability in specifying and maintaining requirements.

There are still few quantitative approaches that successfully capture requirements evolution [1, 37, 38, 82]. This affects also our ability in monitoring changes as well as our ability in estimating the cost of changes [8, 9, 11]. The main limit of quantitative approaches are due to the unclear definition of requirements evolution. Moreover there is still little experience in long term monitoring of requirements evolution. This is because it is difficult to capture evolutionary information and to analyse them.

Most of the models used in practice fail to represent hybrid systems (or

socio-technical systems) involving human resources. This limits our ability to integrate behavioural models of the relation between human and computer systems. Social models (e.g., Social Learning [96]) of the human-computer negotiation may enhance our ability in designing socio-technical systems. Unfortunately the link between technical evolution and socio evolution of socio-technical systems has not yet been fully registered and still remains vaguely understood. Hence there are little models that enable us to fully specify evolution of socio-technical systems. Thus we need to capture evolution of socio-technical systems by collections of models providing us with the problem of integrating and relating them each other.

Dependability related models capture in different ways evolution. Fault tolerance models [51, 70] are based in terms of the distribution , occurrences of failures and interval between them (e.g., Mean Time Between Failures) in computer-based systems. Monitoring these type of measure allows to characterise the evolution of system properties (e.g., Reliability, Availability, etc.). Probabilistic models [60] are used to predict how dependability related measures may evolve according to the estimations of attributes and the assumptions about the operational profile of the system. Whereas models assess system dependability, other models relate it to the structure of the system and its development process (e.g., [58, 59]). Thus it is possible to relate failures profiles to design attributes (e.g., diversity) and system structures (e.g., redundancy). Structured models (e.g., FMEA, HAZOP, FTA) assess the hazard related to system failures and risk associated with them [91]. These models are used to specify safety requirements for safety-critical systems. Despite that these models capture a dynamic view of system failures, they do not capture evolution. Hence they need to be repeated any time evolution occurs, that is, evolution invalidates safety related analysis. There is the need to capture safety aspects on the fly. The cycle of undependability impairment takes into account that undependability is the result of a chain of events and combination of failures. This aspect is also captured in other models analysing the process of failing and the failing structures. The *Domino* model assumes that an accident is the end result of a chain of events in a particular context [34]. Whereas the *Cheese* model consists of different safety layers having evolving undependability holes. Hence system failures in order to completely arise and becoming catastrophically unrecoverable need to propagate through all the safety layers in place [71].

2.3 Discussion and Remarks

The previous sections point out the variety of evolution for computer-based systems. The way each different evolution contributes to dependability may differ from case to case. Table 2.1 summarises the dependability perspectives related to each different type evolution. This analysis of evolution with respect to dependability allows to refine the definition of dependability taking into account evolutionary aspects.

Definition 1 (Evolutionary Dependability) *(Un)Dependability is that* ***emer-***

Table 2.1: Dependability perspectives of Evolution.

Evolution	**Dependability Perspective**
Software Evolution	Software evolution can affect dependability attributes (e.g., Reliability). Nevertheless software evolution can improve dependability attributes by faults removal and maintenance to satisfy new arising requirements.
Architecture Evolution	Architecture evolution is usually an expensive phenomenon. It does not affect directly dependability, but there is high risk if the evolution process is unclear and little understood. Architecture evolution may be needed to support specific system properties (e.g., redundancy, performance, etc.).
Requirements Evolution	Requirements evolution does not directly affect dependability, but non-effective management of the requirement process may allow undesired changes to fall down into the product affecting its dependability. On the other hand requirements evolution may enhance system dependability across subsequent releases.
System Evolution	System evolution may give rise to undependability. This is due to incomplete evolution of system resources. Evolution of some resources (e.g., software) should be taken into account by the other resources (e.g., liveware and hardware) in order to register a new configuration for the system. Hence the interactions among resources serve to effectively deploy a new system configuration.
Human (-Computer) Evolution	Human can react and learn how to deal with undependable situations, but continuous changes in the system configuration may give rise to little understanding about the system. Hence the human-computer interaction may become quite undependable as well.
Organization Evolution	Organization evolution should reflect system evolution. Little coordination between system evolution and organization evolution may give rise to undependability.

gent *property of a computer-based systems such that reliance* ***eventually*** *can justifiably be placed* ***to some extent*** *on the service it delivers.*

In practice all the evolutions interact each other. The interactions of the different evolutions depend on the particular context implementing mechanisms to propagate (or filter) changes. The different evolutions identified allow us to define a notion of evolution for computer-based systems.

Definition 2 (Evolution) *Evolution of computer-based systems consists of the different (meta-)evolutions together with the (macro-)interactions among them.*

The combination of the above definitions allow us to define the concept of *Dependable Evolution* for computer-based systems.

Definition 3 (Dependable Evolution) *(Un)Dependable Evolution consists of those (meta-)evolutions and the set of (macro-)interactions among them that eventually provide to a certain extent an emergent (un)dependability of the evolving computer-based system.*

This taxonomy points out practical issues about evolution. There are many different assumptions about evolution embedded into methodologies. The different understandings of evolution may result to be inconsistent all together. This emphasises the need to related to some extent the different evolutions. The relation of the different evolutions may be useful to link different evolutionary modelling. Little coordination may give rise to undependability in place. Our definition schematises the analysis of evolution. How to classify evolution in context needs to be supported by empirical evidence. Unfortunately evolutionary data are difficult to capture and analyse. Data are often incomplete, distributed, unrelated and vaguely understood. There is the need to identify empirical framework [4] to analyse evolution in context.

This taxonomy is a starting point for practical challenges. Any computer-based system related environment can be classified according to the evolution (defined in terms of meta-evolutions and macro-interactions) in place. This may allow to devise empirical models of evolution. The empirical information collected may capture how the different meta-evolutions are linked together by macro-interactions. This may allow to identify repeatable methodologies to design and deploy evolving computer-based systems.

In coclusion, this chapter describes a taxonomy of evolution for computer-based systems. The taxonomy analyses different layers in which evolution takes place and the different models taking into account evolution. This review points out that evolution consists on many different (meta-)evolutions interacting each other. We therefore give a definition of evolution of computer-based systems. The relation between evolution and dependability allows to identify how the different (meta-)evolutions contribute to dependability. The presented taxonomy furthermore clarifies the different viewpoints from which it is possible to analyse evolutionary phenomena. This chapter provides new insights for understanding evolutionary phenomena of computer-based systems. The following chapters provide evolutionary analyses of case studies.

Chapter 3

An Avionics Case Study

This chapter describes an avionics safety-critical industrial case study that we have analysed focusing on the requirements evolution. The case study consists of software that must satisfy the most stringent level of DO178B [77]. The empirical investigation takes into account the software requirements evolution of the case study. There are two main business stakeholders cooperating in the definition of the system's requirements. These are the *customer*, who provides the engine's requirements, and the *supplier*, who produces the software.

3.1 Case Study Features

The description of the avionics case study identifies critical features with respect to evolution. We describe the case study in general terms, because our focus is on methodologies. The following features are identified in the case study.

Safety Requirements. A system safety assessment analyses the system architecture to determine and categorise the failure conditions. Safety related requirements are determined and flowed down to the software and hardware requirements.

Functional and Operational Requirements. The customer provides the system requirements, which contain information needed to describe the functional and operational requirements for the software. This includes timing and memory constraints and accuracy requirements where applicable. Requirements also contain details of inputs and outputs the software has to handle with special reference to those where non-standard data formats are used.

Software Development Process. Figure 3.1 shows the development life cycle. The bulk of the software development task can be split into two broad areas, that of software design and code and the other of verification. Two main elements serve to complicate the situation. Firstly there are feedback loops that

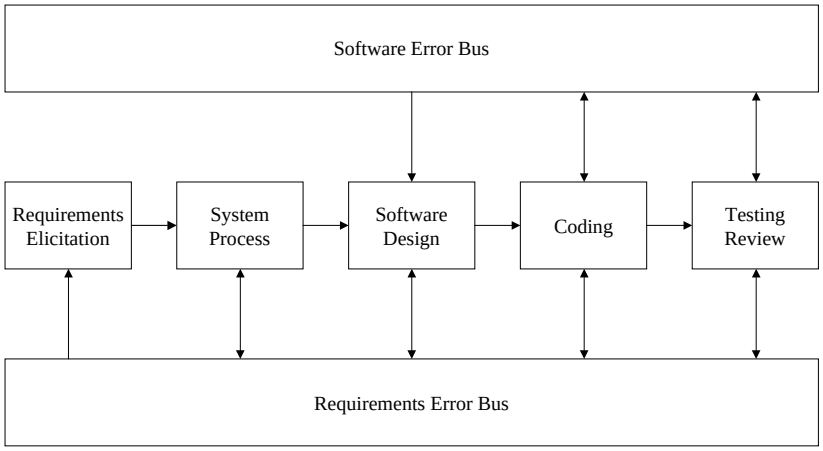

Figure 3.1: The safety-critical software life cycle of the case study.

occur due to problems or modifications. Whenever changes and modifications will arise, they will cause feedback loops into the software life cycle. Notice that as software development progresses from requirements to design and code, problems require items further back in the chain to be modified. The size and scope of these changes will be different for each modification and for each project, thus only general guidelines have been defined from the supplier. For example, while coding a part of the software detailed design a problem may be found with the design or an easier implementation may be constructed which requires the original design to be modified. Also modifications, which are due to problems, hardware changes or even stakeholders, may be introduced from the requirements at any stage of the software project. Secondly there is an expansion of information down the design chain to the code. As design progresses, the software requirements are partitioned into smaller more manageable items. This fragmentation is also reflected into further down activities (e.g., testing) and deliverables (e.g., code). This means that the coding phase at the end of the chain is not a single phase but a collection of interrelated phases. The boundaries between phases are not clearly delineated and some instances overlap. For example, as the code consists of many items it may be possible to begin the integration of some of them before the code for the rest is completed. This is often the case, even on small simple projects. Then it is very difficult to determine the end point of a process and to fix the point in which a project transition

occurs from one phase to another. Thus, elements within each phase may be in different progress stages. There is a strict policy for configuration management, which also requires to maintain traceability on the project to ensure that the final code is complete and consistent with its higher level requirements, and the verification has been performed on the correct standard of the final code. Software verification consists of two main elements testing and review/analysis. At the start of the project, the verification activities have been planed in specific documents detailing the level of verification performed at each phase of the software development. There are two significant milestones that are encountered during a project. One milestone is certification. For modifications to a certified standard of software the full life cycle phase of testing and verification should be complied with. In essence the project should start again, from the beginning, all the phases are repeated. The extent to which all the phases have to be repeated will depend upon the time elapsed since certification. Another milestone occurs around the end of the project life cycle, between the point where the software is being used in flight trials but before certification has been reached. During this stage of the life cycle, new issues to the project specific documents will be required. Also, depending on the size of the modifications, some of the review stages will need to be repeated.

Figure 3.2 shows a representation of the phases of the development process and its deliverables (i.e., system requirements, software functional requirements, etc.). The *System Requirements* cover the whole system, that is, they are specified in terms of system and not software functions. Then the *System Process* translates the requirements and allocates them into the *Software Functional Requirements*. The software functional requirements are organised in terms of the software functions identified in the system requirements. These software functions will be integrated further in the development process. After the definition of the software functional requirements, the development process travels through design and coding. All the anomalies (e.g., faults, failures, misbehaviours, etc.) encountered during the development are reported by a *Fault Report*. Fault reports consist of a document reporting all the information useful to the development team in order to assess the possible faults. When a fault has been recognised actions are taken to fix it and the needed changes are allocated to a specific software release. The scope of these actions ranges from requirements to software code. Thus continuous feedback is provided by the fault reports. Notice that certification requires that all the changes are traced.

Product Line Aspects and Standards. Hardware dependent software requirements arise not from the system requirements directly but from the implementation of those system requirements. This is normally due to the way that the hardware has been designed to meet its requirements and as an indirect result of safety related requirements. The hardware dependent software requirements characterised the specific product-line in terms of hardware constraints and safety requirements. A certification plan for software is the primary means used by the designed authorities to assess whether the software development

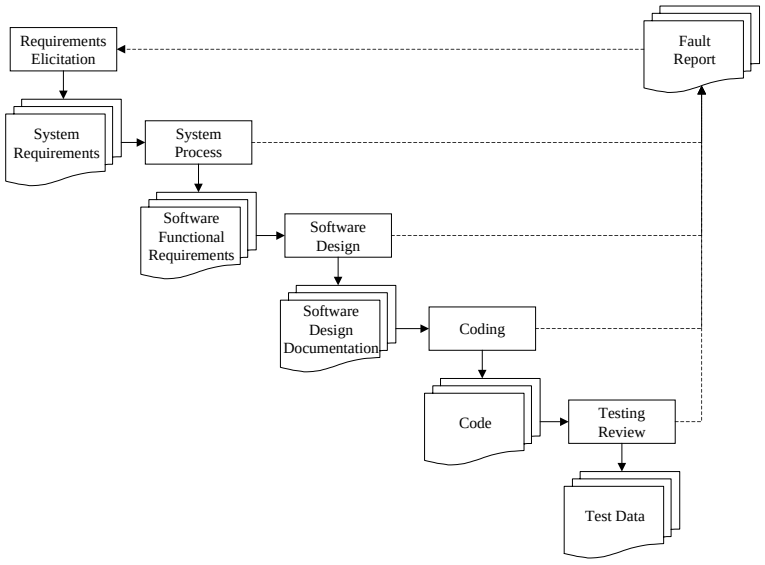

Figure 3.2: Development activities and deliverables.

process proposed is commensurate with the software level proposed. The plan of the case study has been produced according to the specific guidelines in the standard RTCA/DO-178B [77].

Evolution and Maintenance. During the development life cycle, changes and modifications arise which cause feedback loops. As the size and the scope of these changes will be different for each modification and for each project, only general guidelines have been defined. For modifications to a certified standard of software the full life cycle process of testing and reviews should be complied with. The extent to which all activities have to be repeated will depend upon the time elapsed since certification. Modifications that are introduced prior certification will be incorporated during the development life cycle.

3.2 Empirical Analysis

The case study consists of software for an avionics system. There are 22 successive releases of the software requirements specification each corresponding to a software release. The empirical investigation of the avionics case study is based on analyses of data repositories of requirements evolution. The aim is to identify requirements properties [1, 2] that may enhance our ability in controlling Requirements Evolution [1]. The investigation goes from a general viewpoint towards a product-oriented viewpoint. The incremental investigation is summarised in what follows.

3.2.1 General Requirements Evolution.

The initial empirical investigation [1, 2, 4] of the avionics case study is based on analyses of data repositories of requirements evolution. The aim is to identify requirements properties that may enhance our ability in understanding Requirements Evolution. Figure 3.3 shows the total number f requirements changes, i.e., added, deleted and modified requirements, over the 22 software releases[1].

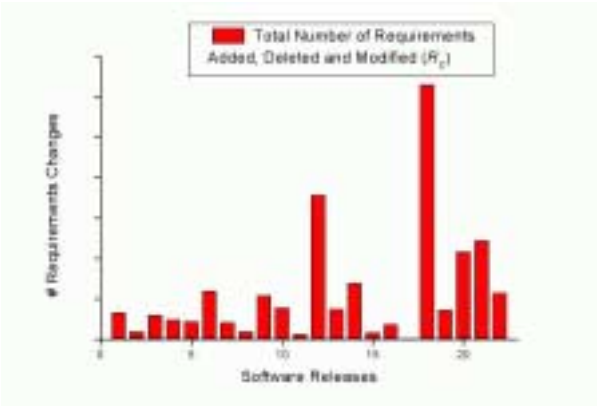

Figure 3.3: Number of requirements changes per software release.

The trend of requirements changes does not give enough information about evolutionary features, but it emphasises Requirements Evolution. The analysis of requirements evolution points out that requirements changes are not uniformly spread out over the three basic changes (i.e., added, deleted and modified requirements), in fact the total number of requirements constantly increases over the software releases. Figure 3.4 shows the increasing number of requirements forming the specification of the avionics system. The increasing trend is due to the fact that requirements become clearer to the stakeholders, who split complex requirements into smaller and more precisely stated requirements. Another reason is that new requirements arise during the progress of the project, because there are requirements that can not be defined at the beginning due to lack of information and requirements dependencies. Finally, design, implementation and testing activities provide additional feedback to the requirements. Thus there is a predominace of added requirements.

[1]There is a correspondence one-to-one between the versions of the requirements specification and the software releases

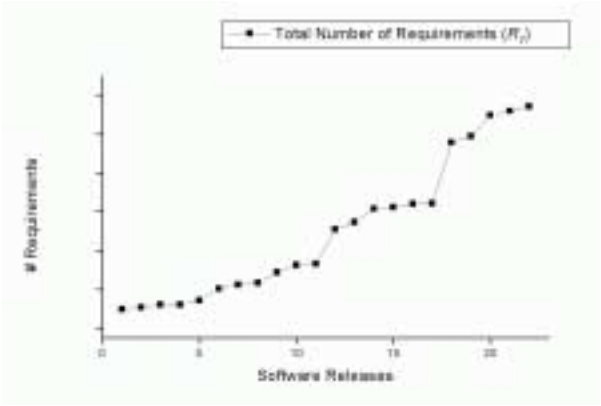

Figure 3.4: Total number of requirements per software release.

3.2.2 Functional Requirements Evolution.

To provide a more detailed analysis our focus moves from the total number of requirements changes to the number of requirements changes in each function that forms the software functional requirements. The software functional requirements fall into 8 functions for which separate documents are maintained. Figure 3.5 shows the trend of the cumulative number of requirements changes for each function. Figure 3.5 points out that the likelihood that changes can occur into specific functions is not constant over the software releases. An outcome of this functional analysis is that the function F1 is not likely to change, therefore it could be considered a stable part of the system. This aspect becomes interesting, because the specific function describes the hardware architecture of the system onto which the software architecture is mapped. It seems furthermore that functions that are likely to change during early software releases change less during later releases, and vice versa. This aspect helps to relate requirements changes with the software life cycle. The different occurrences of requirements changes throughout the life cycle points out some dependencies among functional requirements. Understanding these dependencies may improve the requirements process.

Figure 3.6 shows the scatter plot of number of cumulative requirements changes against the size of each fucntion (at the 22nd release) in terms of number of requirements. Figure 3.6 sows that there exist a linear relationship between the number of changes occuring into a requirements specification and its size. But there exist outliers such as, e.g., F2, F5 and F8.Figure 3.6 moreover points out the intuitive interpretation that the functions above the diagonal dividing the plan are more unstable (in terms of requirements changes) than that ones under the diagonal. In particular F1 turns to be a stable part of the system. This is interesting, because the specific function describes the hardware architecture of the system onto which the software architecture is mapped.

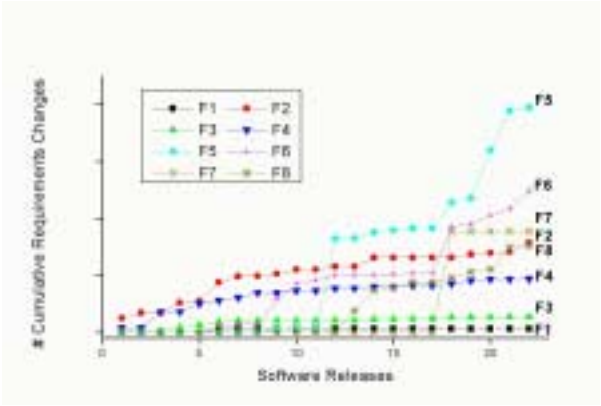

Figure 3.5: Cumulative number of requirements changes for each function

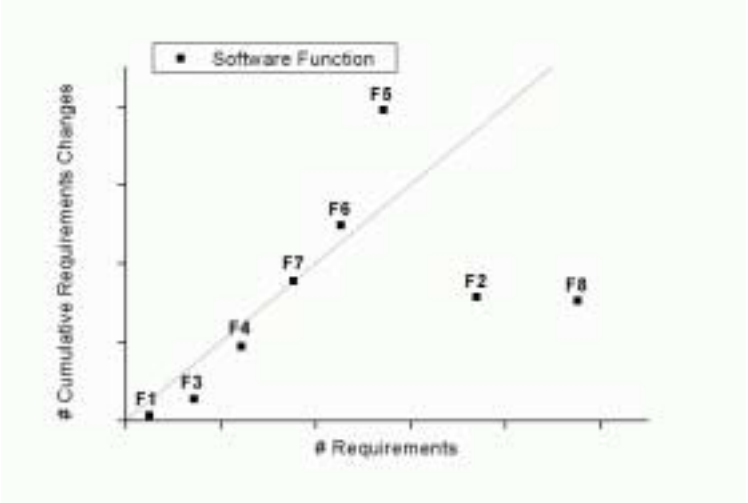

Figure 3.6: Cumulative number of requirements changes against the number of requirements for each function.

The analysis then investigates whether there are differencies among fanctions according to the distribution of changes over the software releases. Figure 3.7 shows the different distribution of requirements changes[2] for all functions. This representation of requirements changes points out how different can be the

[2]Requirements changes are expressed in percetage of the total number of changes occured in the corresponding release

distrubution for each function. The different distributions point out some interesting intuitive properties. Firstly the likelihood that changes can occur into specific functions is not constant over the software releases. Finally it seems that functions that are likely to change during early software releases change less during later releases, and vice versa. This aspect helps to relate requirements changes to the software life cycle. The different occurrences of requirements changes throughout the life cycle points out some dependencies among functional requirements. Understanding these dependencies may improve the requirements process by effective changes allocation and impact analysis.

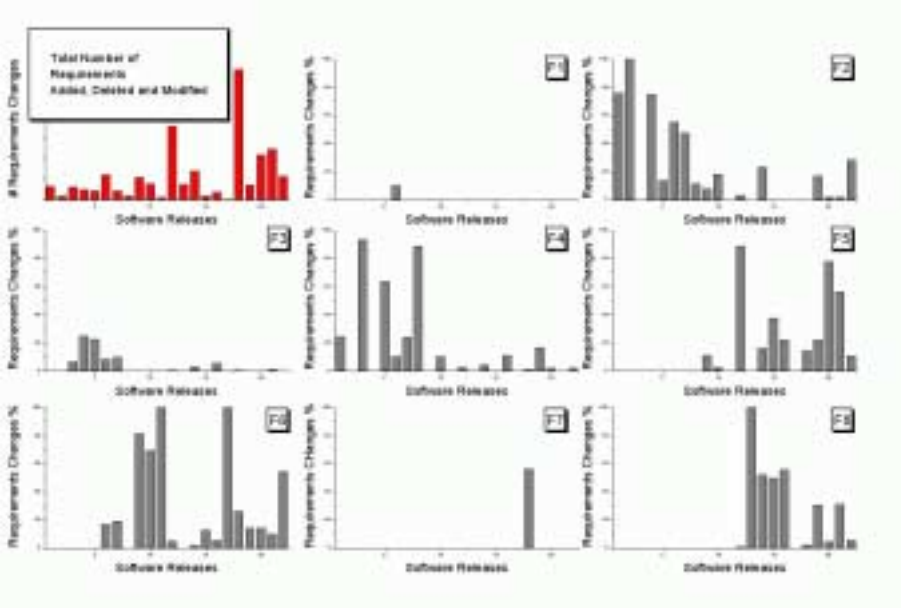

Figure 3.7: Distribution of requirements changes over software releases for each function.

3.2.3 A Taxonomy of Requirements Evolution

The inspection of the history of changes points out the specific changes occurring into requirements. Table 3.1 shows a taxonomy of Requirements Evolution in the case study. Changes affect requirements attributes like variables, functionality, explanation, traceability and dependency. These attributes can be collected and represented by requirements templates (e.g., [76]). These attributes are usually embedded within paragraphs specifying requirements. A structured way of representing, collecting and organising requirements attributes may be useful to identify information for controlling and monitoring requirements evolution. The

taxonomy of Requirements Evolution may help to identify requirements issues. For instance, if changes overlap two categories, the affected requirements may need to be refined in order to fit in one category. This may identify different evolving paths, e.g., splitting requirements in smaller and more detailed requirements or clarify (i.e., modify) their specifications. This is the case when there are requirements that overlap software and hardware. The decision whether to allocate requirements to software or hardware may be delayed till there is a clear understanding of the system. Hence a taxonomy of Requirements Evolution may classify not only an industrial context, but it can be used as a tool during design to identify requirements issues.

Table 3.1: Type of change identified by inspection of the history of changes in the case study.

Type of Change	**Description**
Add, Delete and Modify requirements	Requirements are changed due to the specification process maturity and knowledge.
Explanation	The paragraphs that refer to a specific requirement are changed for clarity.
Rewording	The requirements itself does not change, but it is rephrased for clarity.
Traceability	The traceability links to other deliverables are changed.
Non-compliance	A requirement that is not applicable for a new software package. This is the case when the requirements specification is based on that one of a previous project.
Partial compliance	A requirement that is applicable partially for a new software package. This is the case when the requirements specification is based on that one of a previous project.
Hardware modification	Several changes are due to hardware modifications. This type of change applies usually to hardware dependent software requirements.
Range modification	The range of the variables within the scope of a specific requirements is modified.
Add, Delete, Rename parameters/variables	The variables/parameters to which a specific requirement refers can change.

3.2.4 Requirements Dependencies

The analysis of requirements evolution per software function points out some dependencies between functions. We have evaluated the dependencies between software functions by the number of common fault reports arose during the software life cycle. The dependencies between two functions have been quantified by the number of fault reports overlapping requirements changes. The underlying hypothesis implies that if two functions are modified due to the same fault report, then there are some dependencies between them. Table 3.2 shows the dependencies matrix that has been obtained according the above assumption.

Table 3.2: Requirements dependencies matrix.

F1	**F1**							
F2	2	**F2**						
F3		3	**F3**					
F4	3	1	1	**F4**				
F5	1	4	2	6	**F5**			
F6				1	1	**F6**		
F7				1	1	1	**F7**	
F8	1	4	3	5	9	2		**F8**

We take the number of fault reports in common between two functions as *Dependency Index* between the corresponding requirements. For example, the Dependency Index between the functions F4 and F8 is 5, which means that there have been 5 fault reports in common between F4 and F8. The blank entries in Tab. 3.2 mean that there are not common fault reports between the corresponding crossing functions. The requirements dependencies matrix, Tab. 3.2, gives a practical tool to assess to which extent software functional requirements depend each other. Moreover it identifies those particular fault reports that trigger changes into different functions. Thus an analysis of such identified fault reports may give important information about requirements. In order to provide feedback into the software organisation the matrix may be analysed to assess the ability of the organisation in identifying software functional requirements for a system or a series of systems for a particular product-line. A tool as the dependencies matrix supports software product-line engineering [94]. Future refinements over product-line projects may identify an effective set of modular system functions such that to reduce disturbing dependencies (e.g., Dependency Index equals to 1). Moreover the Dependency Index may be used to refine impact changes estimates based on traceability.

3.2.5 Requirements Maturity Index

Metrics [27] may be used to quantify some properties monitoring requirements evolution. The standard IEEE 982 [37, 38] suggests a *Software Maturity Index* to quantify the readiness of a software product. The Software Maturity Index can be used for software requirements, hence a *Requirements Maturity Index* (RMI) to quantify the readiness of requirements. Equation 3.1 defines the RMI[3].

$$RMI = \frac{R_T - R_C}{R_T} \ . \tag{3.1}$$

[3] R_T is the total number of software requirements in the current release; R_C is the number of software requirements in the current release that are added, deleted or modified from the previous one.

Figure 3.8 shows the RMI calculated for each software release with respect to the total number of requirements (R_T) and the total number of changes (R_C). The RMI results to be sensitive to the requirements changes occuring release after release, but it does not take into account historical information about changes. Thus whenever there is a substantial number of changes (this may be the case of a new release) the RMI reflects the introduction of changes without capturing information related to the number of releases and the average number of changes introduced over previous releases. The RMI therefore captures stepwise (release after release) changes, hence it is too sensitive to changes introduced into a single release overdegradating its assessment about the readeness of requirements.

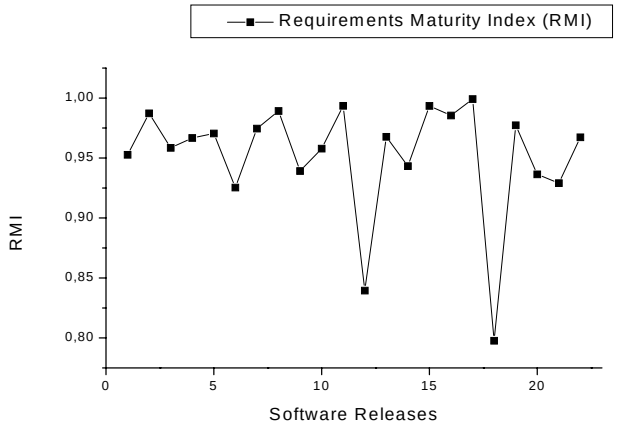

Figure 3.8: Requirements Maturity Index for each software release.

3.3 Requirements Evolution Modelling

The initial investigation provides input to model Requirements Evolution. This section introduces our empirical model of Requirements Evolution and its process. Requirements Evolution is modelled according to three different aspects aiming to classify, structure and quantify evolutionary features.

3.3.1 Structuring Requirements Evolution

The backward reconstruction of the history of changes identify a structured work flow of Requirements Evolution. This structure introduces a graphical model that has been identified by reasoning on data repository of the case study and by inspecting the requirements documents and their respective history of changes. The inspections of the requirements documents and history of changes aimed to reconstruct how the documents have been modified through the software life

cycle by requirements changes. The graphical model shows a representation of Requirements Evolution in terms of requirements changes (i.e., added, deleted and modified requirements) together with a software life cycle perspective. Using a graphical representation is effective at presenting the overall picture of Requirements Evolution. The diagrammatic representation has furthermore a more intuitive structure than any text-based representation and this structure can be used to reflect how requirements are engineered through the entire life cycle. Finally the graphical representation may allow to easily identify similarities over functions and product lines. Requirements Evolution can be expressed in terms of changes allocated to requirements releases. Before representing Requirements Evolution in terms of changes, we need to define how requirements are represented. They are organised in a document or a collection of documents (one per each function) in which all requirements are uniquely identified by a numeric id. Thus a requirements document can be considered as a collection of sets of uniquely identified requirements. Among the set of requirements there is a partial order defined by the numeric id. There are three basic operations to change requirements. Figure 3.9 shows a graphical work flow representation of the three operations, which are define in what follows.

Add: $Add[n]$ introduces a subset of n new contiguous requirements ordered according to their index into the current requirements document.

Delete: $Del[n]$ deletes a subset of n contiguous requirements from the current requirements document.

Modify: $Mod[n]$ modifies a subset of n contiguous requirements into the current requirements document.

Here n represents the size of the subset changed, whereas the specific changed requirements are identified by the pointed subset in the middle of the three. The subset of m modified requirements is not identified by the graphical representation because the Mod operation does not actually change the structure of the requirements document. Notice that even if the graphical representation in Fig. 3.9 is not formal, it implies some syntax and semantic, which are expressed by the different shapes of the trees, the different edges and the order of the leafs (i.e., subsets of requirements).

At this stage the graphical model aims just to intuitively capture the Requirements Evolution Process and its complexity. Figure 3.10 shows the representation of the work flow evolution for the function F1. The shape (even without detailed information) of the requirement evolution captures the complexity of the Requirements Evolution process. The order of the operations is left to right, hence the most left node is the initial set of requirements. The initial set of requirements is then modified by a sequence of basic operations (i.e., Add, Del, Mod). Requirements changes are allocated over different software releases. The graphical representation of requirement evolution for the other

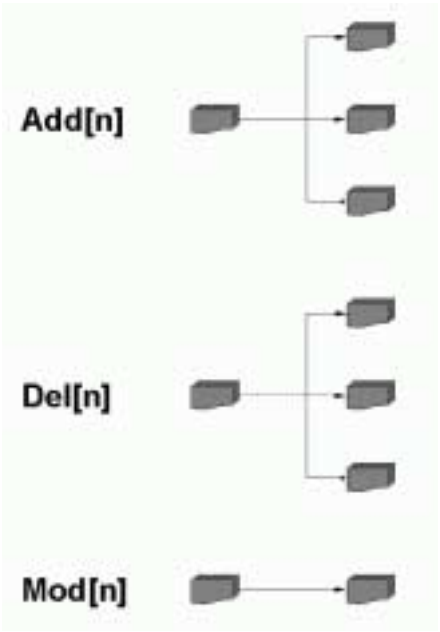

Figure 3.9: Graphical work flow representation of the basic operations to changes requirements.

functions in the case study is more complex that that one, emphasising how the structured work flow can easily capture the Requirements Evolution process. It captures how requirements are changed throughout the software life cycle.

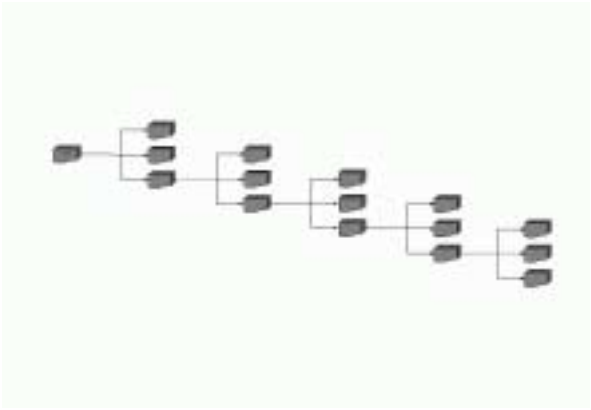

Figure 3.10: Graphical work flow representation of requirement evolution for F1.

3.3.2 Quantitative Requirements Evolution

The empirical analysis of the case study points out that requirements changes are not uniformally distributed over the software releases. There is moreover a different distribution for each function. The behavioural analysis of the RMI on the dataset points out that the RMI is not sensitive to the history of changes. This is because the RMI is defined for two subsequent releases, that is, the RMI does not take into account historical information. For instance, information like how long some changes have been introduced, how many releases there were without changes, etcetera. We propose to refine the RMI by taking into account historical evolutionary information and not just the requirements changes occurred in the next subsequent release. The simplest historical information consists of the *Cumulative Number of Requirements Changes* (CR_C) and the *Average Number of Requirements Changes* (AR_C), Eq. 3.2, over the software releases.

$$AR_C = \frac{CR_C}{n} \ . \tag{3.2}$$

Based on the above information we propose two refinements of the RMI. The first refinement is named *Requirements Stability Index* (RSI) , Eq. 3.3. Differently from the RMI the RSI takes into account the cumulative number of requirements changes, CR_C. Hence RSI is sensitive not just to the total number of requirements, R_T, but also to the cumulative number of changes, CR_C. The RSI can be also negative in cases where there have been more changes than the total number of requirements, R_T. The maximum value of SRI is 1 in the case when there have not been changes since the initial release (this is quite unlikely in practice).

$$RSI = \frac{R_T - CR_C}{R_T} \ . \tag{3.3}$$

The second refinement is named *Historical Requirements Maturity Index* ($HRMI$), Eq. 3.4. It is defined on the total number of requirements and the average distribution of requirements changes over the software releases (AR_C). This allows to take into account the age of changes.

$$HRMI = \frac{R_T - AR_C}{R_T} \ . \tag{3.4}$$

The intuition behind the two refinements is that if some changes are introduced into requirements, these changes affect the stability of the requirements. The maturity of the requirements depends on how long some changes have been introduced. That is, if there are not further changes the maturity gradually increases with the delivery of subsequent releases. Figure 3.11 shows the simulation of the above metrics on a sample case. The top, middle and bottom picture of Fig. 3.11 respectively show the requirements changes, the total number of requirements and the comparison of the RMI, RSI and $HRMI$. The

comparison of the three indexes, i.e., RMI, RSI and $HRMI$, shows clearly the different behaviour with respect to the sample scenario. The $HRMI$ is less sensitive and more stable than the RMI, which is more sensitive to the number of changes occurring release after release.

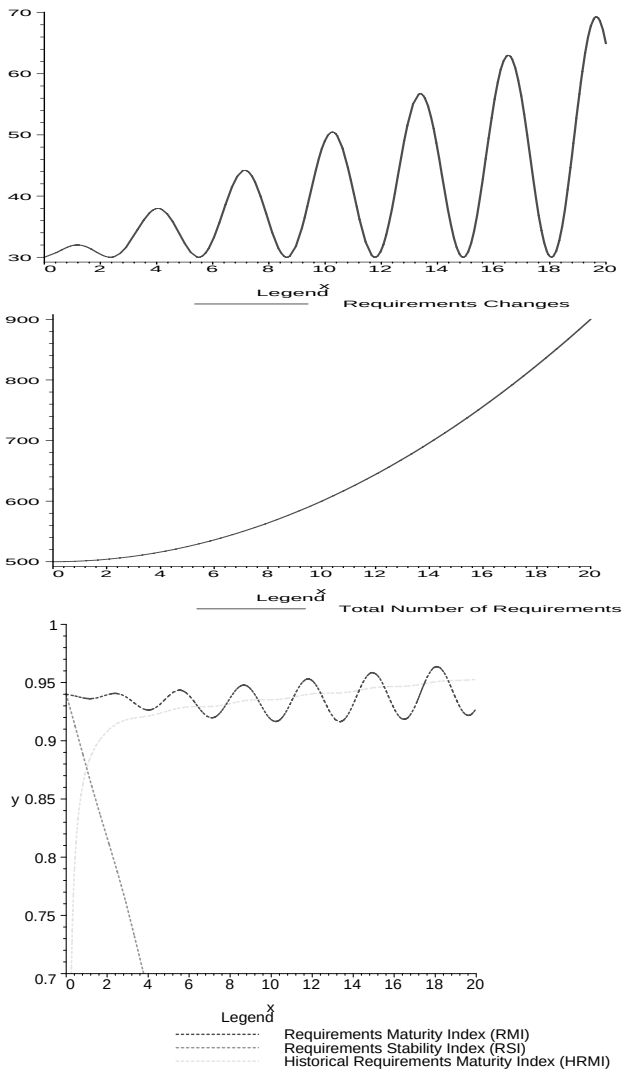

Figure 3.11: Simulation of the requirements evolution metrics on a sample scenario.

3.3.3 Measuring Requirements Evolution

This section assesses on the case study the proposed quantitative models of Requirements Evolution. Figure 3.12 shows the average number of changes over the software releases. The picture shows clearly an increasing trend. This is particular for the analysed case study, but there could be different distribution according to the specific system, the adopted design process and the system life cycle.

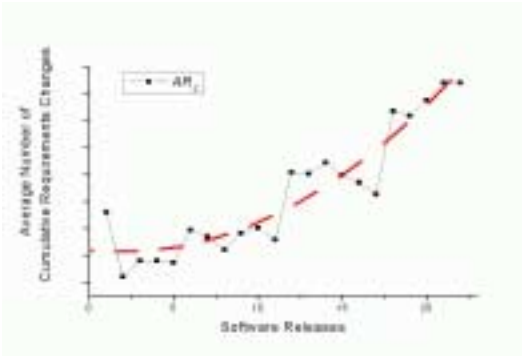

Figure 3.12: Average number of requirements changes.

As we would expect the RSI has a decreasing trend. Whereas it is more interesting to analyse the RSI at the last release (i.e., the 22nd release) of the case study. Figure 3.13 shows that the RSI captures the stability of each function. Those functions with negative RSI are the most unstable in the system. The RSI can be easily applied to any dataset, because it takes into account not just the number of changes but also the dimension of the dataset considered. Hence the RSI estimates how stable is a set of requirements.

Figure 3.14 shows the $HRMI$ for the entire set of requirements. The $HRMI$ smoothly follows the iterations of the development life cycle. That is,decreasing trends are associated to the introduction of major changes (establishing a new major release), which trigger a stabilisation process into subsequent releases during which the $HRMI$ increases again. $HRMI$ is less sensitive that RMI to changes over single releases, but it smoothly captures the requirements process and its stabilization.

In conclusion this section assesses the proposed metrics, RSI and $HRMI$. The analysis supports the intuition behind the two metrics. That is, RSI evaluates the overall stability of a set of requirements and $HRMI$ evaluates the

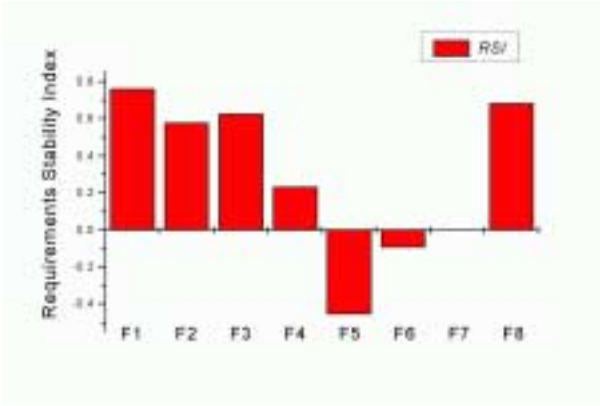

Figure 3.13: Requirements Stability Index for each function at the 22nd release of the requirements specification.

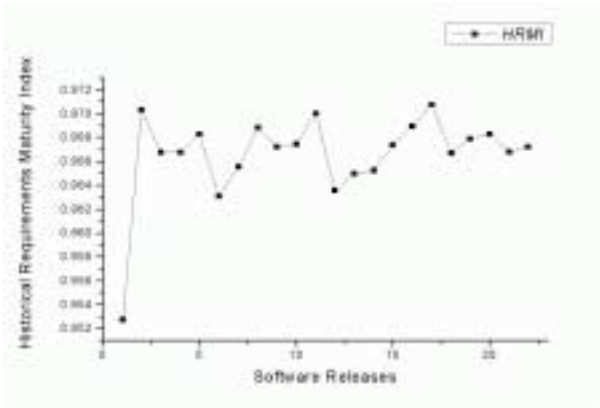

Figure 3.14: Historical Requirements Maturity Index for the entire set of requirements.

requirements maturity with respect to the requirements evolution process. This points out that the distribution of the requirements changes is an importan driver for the RSI and the $HRMI$. This final result relates the Requirements Evolution process to the development process and provides an attribute to control the maturity of requirements. Different distributions imply different trends of RSI and $HRMI$.

Chapter 4

A Smart Card Case Study

This chapter describes a smart card industrial case study that we have analysed focusing on requirements viewpoints and organisation. The case study consists of an industrial context for the production ans support of smart card systems. The investigation takes into account mainly organisational issues related to requirements viewpoints.

4.1 Case Study Features

People use smart card systems in their daily life. Credit cards, Pay-TV systems and GSM cards are some examples of smart card systems. Smart card systems provide interesting case studies of distributed interacting computer-based systems. Behind a simple smart card there is a complex distributed socio-technical infrastructure. Smart card systems are

Real Time Systems. The transactions of smart card systems occur in real time with most of the services operating on a 24-hour basis. The availability of smart card systems is fundamental to support business (e.g., e-money) and obtain customer satisfaction.

Interactive Systems. Most of smart card systems operate on demand. The request of any service provided depends on almost random human factors. Operational profiles show that human-computer interaction turns to be one of the critical factors for smart card systems.

Security Systems. Smart card systems often manage confidential information (e.g., bank account, personal information, phone credit, etc.) that need to be protected from malicious attacks.

The considered smart card context is certified according to many quality and security standards. In particular its management process is conform to

PRINCE2. PRINCE[1] (PRojects IN Controlled Environments) is a structured method for project management [14]. It is used extensively by the UK Government and is widely recognised and used in the private sector, both in the UK and internationally. PRINCE2 is a process-based project management approach integrating a product-based project planning. The investigation of the smart card context aims to identify general aspects in requirements management. We do not intend neither to validate any particular methodology nor to assess the specific industrial context. Our aim is to identify general practical aspects that can improve our ability in dealing with Requirements Evolution.

4.2 Requirements Viewpoints

The analysis of the smart card case study focuses on requirements viewpoints [87]. Viewpoints provide different perspectives to manage requirements evolution. The case study provides the opportunity to identify hierarchical viewpoints within the organisation to which correspond different levels of management and different requirements. Viewpoints analysis furthermore points out both process and product divergencies, which can be used for further investigations. The investigation analysis is based on interviews and questionnaires [3]. The analysis of the smart card organisation points out three different viewpoints named *Business*, *Process* and *Product Viewpoints*. They correspond to different management levels and responsibilities within the organisation. At each level corresponds different processes and different requirements. All the three viewpoints together contribute to the overall management of requirements evolution. The following subsections describe the three viewpoints and their respective processes.

4.2.1 Business Viewpoint

The business viewpoint is associated to an high management level within the organisation. This level is where project are originated and the interaction between the organisation and customers takes part. Figure 4.1 shows the business process of the organisation. A customer interested in one of the smart card systems provides requirements that are integrated together with that general ones of the system. Customer requirements aim to complete the smart card system requirements. The requirements are then negotiated between the customer and the bureau, that is, the department responsible for the spin out of smart card projects. It provides production constrains (i.e., additional requirements). When Customer and Bureau agree on the system requirements the project is declared LIVE, that is, from this moment onward the production of the smart cards starts. The production is organised in terms of subsequent deliveries. During the production of each delivery new requirements may arise due to feedback from users of the new smart cards (e.g., misbehaves, request

[1] PRINCE is a registered trademark of CCTA (Central Computer and Telecommunications Agency).

of new services, etc.) or to business constraints (e.g., production issues, request of additional cards, etc.). The business viewpoint therefore deals with the management of system and business requirements which are not directly related to software requirements. The main activities in the business process, Fig. 4.1, consist of many sub-activities, which are not visible at the business level. Once requirements change the management passes to a sublevel, namely Process, identifying another viewpoint.

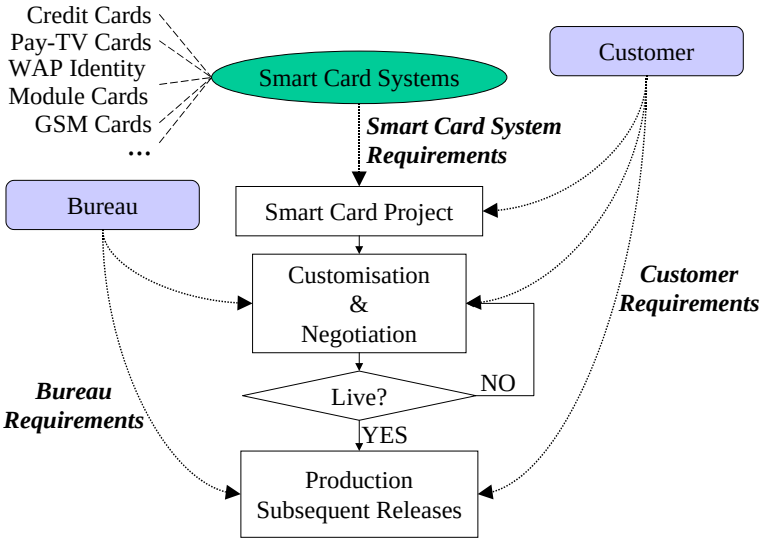

Figure 4.1: Business process.

4.2.2 Process Viewpoint

An intermediate level within the organisation is represented by all the organisation's management processes. The process viewpoint consist of all the management processes adopted within the organisation [14]. Figure 4.2 shows the process to manage requirements changes. Every time a request of change is raised the process for changes management starts. The initial part of the changes management process is a macro-process of the negotiation activity. If changes require some software development a set of analyses is performed. Each of these analyses corresponds to a different subsystem consisting of a part of the whole smart card system. For each subsystem it is produced an impact report, which also estimates the cost of changes in terms of man-day. The set of impact reports serves as basis for the negotiation of changes. The agreed software changes are given as input to the software development development life cycle, which represents the gate to a another viewpoint, namely Product, in the organisation.

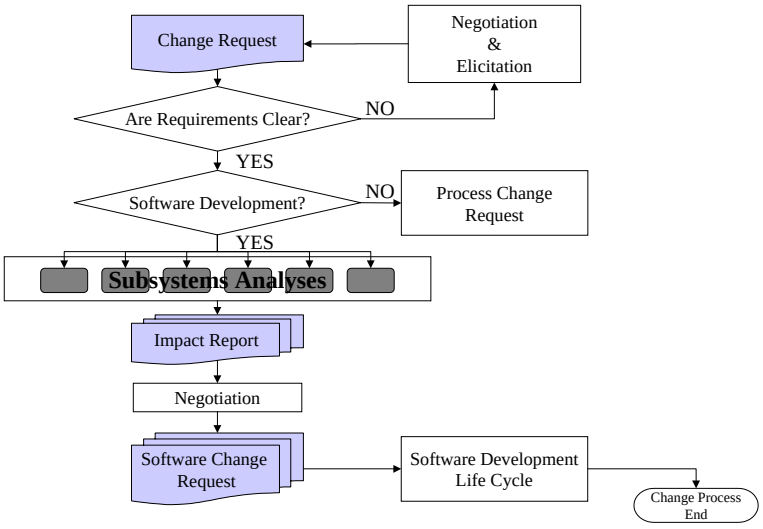

Figure 4.2: The process to manage requirements changes.

4.2.3 Product Viewpoint.

The software level identifies another viewpoint named Product. Here the software follows its own development process. Software requirements are elaborated through the process and requirements changes are allocated to subsequent releases of the software behind the smart card system. Figure 4.3 shows the software development process, which is a V model [67]. At this level software changes are drawn down through the development process. Differently from the Process viewpoint that estimates changes in terms of man-day, the Product viewpoint takes into account software changes in terms of activities (e.g., coding and testing).

4.3 Viewpoints Analysis

Requirements Engineering viewpoints have been also investigated by a questionnaire [3] to assess the general understanding of the requirements process within the organisation. The questionnaire consists of 152 questions grouped according the categories in Fig. 4.4. People with different responsibilities within the organisation filled in the questionnaire. Figure 4.5 shows the profiles of the questionnaire for two persons with similar experience and with different responsibilities within the organisation. They correspond to two different management levels within the organisation and are respectively associated to the Product and Process viewpoints. The questionnaire captures how the requirements process is interpreted from different viewpoints within the organisation.

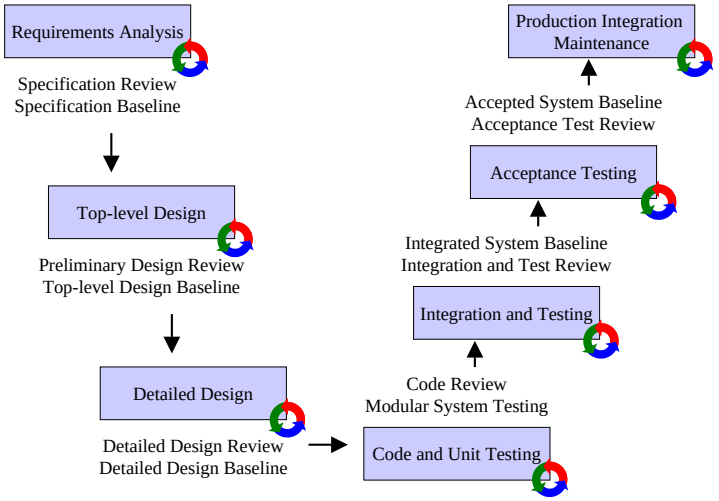

Figure 4.3: The software development process: V model.

1. Requirements Management Compliance
2. Business Tolerance Requirements
3. Business Performance Requirements
4. Requirements Elicitation
5. Requirements Analysis Negotiation
6. Requirements Validation
7. Requirements Management
8. Requirements Evolution & Maintenance
9. Requirements Process Deliverables
10. Requirements Description
11. System Modelling
12. Functional Requirements
13. Non Functional Requirements
14. Portability Requirements
15. System Interface
16. Requirements Viewpoints
17. Product-Line Requirements
18. Failure Impact Requirements

Figure 4.4: The groups of requirements engineering questions.

Figure 4.6 compares two similar viewpoints (i.e., the viewpoint associated to the project manager) with a third viewpoint (i.e., the viewpoint associated to the software development manager) to take into account some bias in the analysis. The similar viewpoints have similar trends. Whereas there are some divergencies between the different viewpoints. The largest ones are those associated with the groups of questions 2, 3, 8, 15, 16 and 17 (see Fig. 4.4). The groups 2 and 3 identifies business aspects of the requirements process. The distance between the answers to the questions in the group 8 points out that there are different level of confidence in the management of requirement evolution. This is probably because the management of changes takes into account process aspects that better fit the process viewpoint. This is also due to some issues in transmitting requirements changes through the management hierarchy within the organisation. The groups 15, 16 and 17 identify product-oriented questions.

Figure 4.5: Comparison of two different Requirements Engineering viewpoints.

The divergency might be due to the fact that the two viewpoints deal with different requirements. The process viewpoint deal with system requirements, which are different from the software requirements taken into account at the product level. The software embedded in a smart card system represent one aspect, which is not completely visible at the process level. These divergencies identify product-oriented refinements for further investigations. They furthermore represent awareness for the organisation, issues arising from these divergencies may cause undependable software and process degradation.

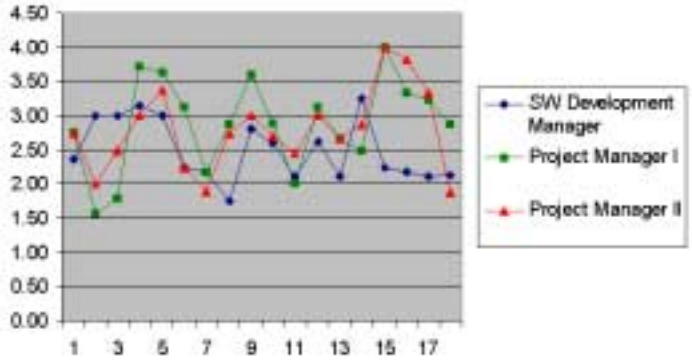

Figure 4.6: Comparison of viewpoints.

4.4 Discussion and Remarks

The smart card case study gives rise to discussions. The empirical analysis points out three different viewpoints within the organisation. We have named these *Business*, *Process* and *Product Viewpoints*. Each viewpoint deals with requirements and their evolution. Requirements management follows a hierarchical structure defined within the organisation. The three viewpoints identified within the organisation represent an hierarchy through requirements evolution expands. This hierarchy identifies different types of requirements and their granularity. Despite this hierarchical structure current methodologies in requirements engineering flatly capture requirements viewpoints [87]. The interviews point out to seek different abilities to support different viewpoints. Different responsibilities seek different types of support. The business viewpoint needs to support project visibility. Most of the process-oriented methodologies in Software Engineering allow to better plan project activities, But they do not completely clarify the link between software features and project activities. This motivates a shift from process to product-oriented software engineering [94]. The process viewpoint seeks support to enhance its management ability by measuring (requirements) evolution. The management process registers requirements changes, but a quantitative approach to measure requirements evolution needs to be organised within the specific environment. The product viewpoint would like to enhance its ability in identifying reusable (product-line) functions and repeatable processes to allocate functions to smart card system requirements. At this level there exist two different opponent processes. Good practice in Software Engineering requires that requirements are divided into smaller and more manageable items. This triggers an information flow expansion throughout the development process. On the other hand specific functions would be allocated to software requirements according also to past experience. The gap between these two processes represents the extent to which an organisation is able to identify an optimal and effective set of software functions. The smaller the gap the better the ability in reusing software functions and identifying product-line features.

Remark 1 *Requirements viewpoints should capture organisation's hierarchy and its granularity.*

Viewpoints therefore need to be supported by providing information that can improve project management practice. This is a crucial aspect in particular in those context characterised by evolutionary development [31]. The interviews point out to seek different abilities to support different viewpoints. Different responsibilities seek different types of support. The business viewpoint needs to support project visibility. Our impression is that most of the process oriented methodologies in Software Engineering allow to better plan project activities. But on the other hand they do not allow to effectivelly link software features to project activities. Then the ability of addressing software characteristics (e.g., reliability) by project activities results to be non effective and optimal. This is one reason to seek for a shift from process to product engineering [94].

The process viewpoint seeks support to enhance its management ability by measuring (requirements) evolution. The management process registers requirements changes, but a quantitative approach to measure requirements evolution needs to be organised within the specific environment. The use of general metrics for requirements evolution can be misleading [1].

The product viewpoint would like to enhance its ability in identifying software function to support reusable functions and a repeatable software process. Moreover this ability will provide a feedback system[2] across projects. At this level there exist two different opponent processes, Fig. 4.7. Good practice in Software Engineering requires that requirements are divided into smaller and more manageable items. This triggers an information flow expansion throughout the development process, Fig. 4.3. On the other hand specific functions would be allocated to software requirements according also to past experience. The gap between these two processes represents the extent to which an organisation is able to identify an optimal and effective set of software functions. The smaller the gap the better the ability in reusing software functions and identifying product-line features. Moreover the set of software functions should identify a suitable trade off to support even function compositionality.

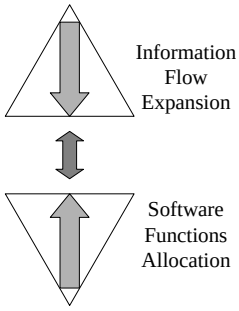

Figure 4.7: Opponent processes within the Product Viewpoint

Remark 2 *Viewpoints within organisations needs to be supported by different abilities (e.g., visibility, measurability and functionality).*

Remark 3 *Engineering methodologies need to be calibrated within an organisation in order to set a trade off supporting good practice management and design. Moreover the identification of a suitable set of functions turns to be crucial to support reuse, compositionality and cross-projects orthogonal classification.*

[2] A feedback system may help to identify a classification for software functions, e.g., old customer - old functions, old customer - new functions, new customer - old functions and new customer - new functions.

In order to improve our ability of dealing with requirements evolution requirements management should take into account particular product aspects. The product viewpoint identifies in the architecture the most stable parts of a smart card system. This aspect, even if it has not been investigated in any specific project of the industrial context, agrees on empirical results of previous work within a different industrial context [1]. Moreover in the smart card case study the architecture implements most of the security requirements. The fact that the architecture implements those requirements that are crucial for the business suggest that stable requirements are associated to those requirements that are vital for business survivability and image. This aspect is also supported from the PROTEUS [68] classification of stable requirements, which are originated in the technical core of the business.

Remark 4 *The architecture represents a stable part of a system and implements those requirements that find origin in the technical core of the business.*

Chapter 5

A Modelling Case Study

This chapter explores the limitations of one technique for modelling computer-based systems with evolving requirements. A case study is introduced which highlights the importance of taking a multi-perspective on dependable computer-based systems. This should be reflected in the modelling technique. Such considerations motivate our ongoing research agenda.

5.1 Case Study Features

ParcelCall[1] is a project looking at creating a *parcel localisation system*: an open distributed system which is to be integrated with the legacy systems of transport and logistic companies (referred to as carriers). The project consortium includes hardware providers, integrators and carriers. ParcelCall, as described below, is an example of how computer-based (or, software) systems are changing in the modern electronic-mediated society. ParcelCall is useful to reveal engineering aspects of computer-based systems as well as general operational failures related to the nature of computer-based systems.

Current demands have transformed the transport and logistics process of today into an increasingly complex process requiring intelligent systems for sorting, planning, and routing; enabling faster and more reliable transportation, while supporting additional services such as time-sensitive deliveries and tracing of products. While many larger companies have developed solutions to provide their customers with such services, high costs impede smaller companies to do so as well. Current services provided to customers include notifications of product delivery or dispatch but are not commonly very precise. The ParcelCall project explores the development of a new low cost information infrastructure that enables the continuous information about the exact geographic position of parcels at any time. Transportation companies will thus be able to offer an additional valuable service to customers: the position and status of transportation

[1] ParcelCall, EU project within the IST programme of the Fifth Framework. Project publications and description can be found at `http://www.parcelcall.com`.

goods can be queried at any time.

Figure 5.1 shows the architecture of the ParcelCall system with three main components:

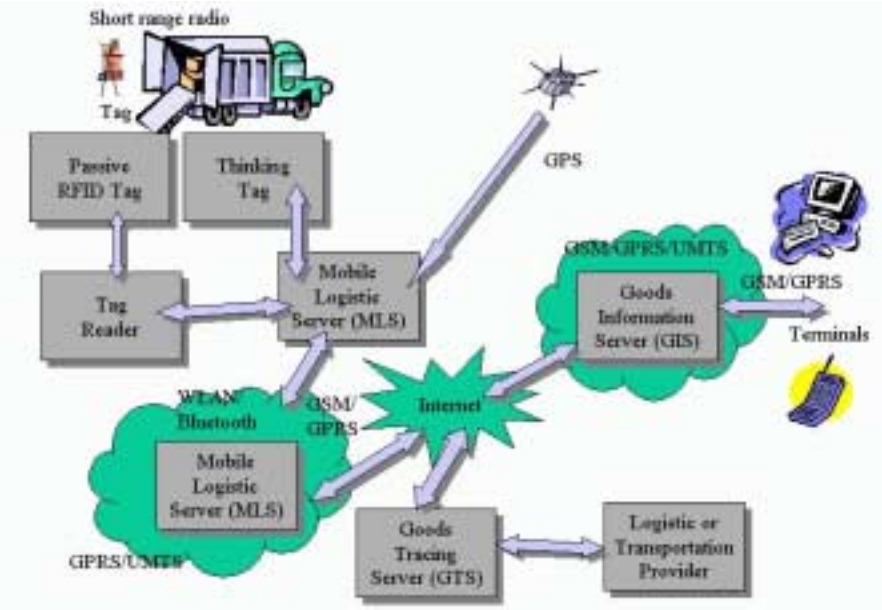

Figure 5.1: The ParcelCall Architecture.

- a *Mobile Logistic Server* (MLS): is an exchange point or a transport unit (container, trailer, freight wagon, etc). The transport units carry the parcels. A parcel has a tag identifying it: a passive radio-frequency identifier (RFID) tag or an active "thinking" tag. A thinking tag is able, for instance, to monitor environment conditions and emit alarms in case certain constraints are exceeded. Since containers can be inside other containers MLSs form a hierarchy. MLSs always know their current location via the GPS satellite positioning system.

- a *Goods Tracing Server* (GTS): comprises several databases one of which contains MLS hierarchies. Moreover, it keeps track of all the parcels registered in the ParcelCall system. GTS is integrated with the legacy system of transport or logistic companies.

- the *Goods Information Server* (GIS): interacts with the customers, provides the authorised customer the current location of her/his parcels, and keeps her/him informed in case of delivery delays. Interactions between the customers and the GIS are bidirectional and done via a terminal or a mobile phone.

All components interact through common open interfaces on top of standardised communication protocols (e.g., TCP/IP, GMS, GPRS, Bluetooth). The legacy system of a carrier is a computer-based system including both human carriers and machines.

The ParcelCall architecture is an example of how the modern electronic-mediated society has evolved since the utilisation of software as a means to perform computations. The original role of software was to automatically execute computations in order to simplify human workload (e.g, mathematical computations). Software consisted of single-platform (almost single-user) code developed and executed in the same environment. Nowadays computer-based systems like ParcelCall are heterogeneous multi-user, multi-platforms and multi-environments distributed systems supporting a wide range of human activities (e.g., computation, safety and security assurance, monitoring, etc.). The wide utilisation of software and its integration in many different applications, ranging from commercial to safety-critical systems, has changed the nature of software. This role revolution has changed the perception of design of computer-based systems [63]. Computer-based systems are not only a means to perform computational activities but also where the computations take place. The computational activity is distributed over the heterogenous resources (e.g., Software, Hardware and Liveware [22]) of a system and may be human as well. Hence cognition is not limited neither to the human cognition nor to the computer computation. The human is part of a computer-based system, and computations are the result of interactions among computer-based activities creating artefacts.

For instance, once integrated with the carrier system, the parcel localisation system will affect and influence the carrier's organisation in several different ways: economically, sociologically, and so on. Moreover, the impact on the work performed by human carriers is particularly important. Notice that in the carrier system computer-based activities consist of entwined human and machine actions, which will necessarily change even if the result of one such activity (e.g., delivery of a parcel) might apparently be the same. The ParcelCall system provides further artefacts, like the response to an alarm emitted by a parcel with a "thinking" tag. This artefact is the resulting outcome of combining human and machine artefacts.

A further implication of the ParcelCall's localisation system concerns quality of service (QoS) requirements like, for instance, timeliness, capacity, predictability, reliability, safety and security. One example of such a requirement in the ParcelCall system corresponds to the time required *normally* for accessing the status and localisation information of an item. According to ParcelCall documents, this should not exceed 15 seconds. This type of requirement is difficult to assess during design (e.g., how to test the system's performance during design?), as it may be sensitive to changeable human and technical factors. Other issues arise with conflicting requirements. For instance, information may be delayed for security in order to protect the parcel from malicious intentions. This delay is a trade-off between performance requirements related to information retrieval (to the customer) and security constrains (to protect the customers' goods). This shows how complex could be the identification and definition of

non-functional requirements (e.g., quality, dependability, survivability, etc.) for computer-based systems.

5.2 Limits

In this section we discuss some issues arising in the specification and design of ParcelCall. We focus on the limits experienced in modelling ParcelCall and in analysing its evolution.

5.2.1 Modelling

As described in the previous section, ParcelCall is a computer-based system consisting of distributed and interacting heterogeneous resources. To model the ParcelCall system and its integration with the carrier system we take a component-based approach. We essentially use a pragmatic extension of UML [65, 78] for component-based design as given in [15]. We have used it to specify some aspects of ParcelCall in [46].

Using UML we assume that the computer-based system under development can be modelled according to the object-oriented paradigm, that is, in terms of objects, relationships between objects, and so on. Furthermore, a component-based approach for specification allows us to model ParcelCall's architecture at a high level of abstraction focusing on the main components and their interactions. UML is mainly a diagrammatic language offering several diagrams to capture different aspects of a system. It includes the Object Constraint Language (OCL), a textual notation for representing static constraints on the model which cannot be given through the diagrams. To adopt the UML modelling language, UML tools and methodologies for the analysis and design of dependable computer-based systems, it is necessary to adapt these to an inter-disciplinary approach.

UML is mainly used by software engineers or computer scientists, i.e., by people who can be easily trained to use it. To develop complex systems which involve a team of people of different disciplines, UML is not adequate as a common language. Even though UML as used in the requirement analysis relies mainly on simpler diagrams (like use case diagrams, sequence diagrams and possibly also statechart diagrams) that should be understandable enough for discussions with non-technical people this is far from being the case. These diagrams have been developed by software engineers (widely used to thinking in object-oriented terms) for software engineers. The diagrams do not reflect different perspectives or understandings of a system or its requirements, ways of working or thinking. Hence using UML implies adopting the design viewpoints represented by the UML syntax. Making available new alternative diagrams developed by people from other disciplines for capturing their understanding of the requirements, and combining their use with the UML diagrams would be a partial solution. A need for integrating these diagrams would follow. Alternatively, more powerful and different kinds of tools should be made available for

a mixed team of people including tools for non-technical as well as for technical people.

From the software engineer's point of view, dependable systems raise further issues which need to be dealt with. In particular, how to capture QoS requirements. UML does not provide an adequate solution to this problem. In the ParcelCall example, for modelling adequately the dependable integration and interaction of the parcel localisation system and carrier system we need to:

- model QoS requirements that may cut across several components in the system;
- model part of the environment of the system as well;
- understand human behaviour and their understanding of the computer-based system.

UML describes the interactions between the software system and its environment (which includes the human) through use cases (which can later be refined into other types of diagrams). There is, however, a strong distinction between the software system and its external environment. There is no intention of integrating the environment, or part of it, into the model. This clearly limits the expressiveness of modelling languages like UML for the computer-based systems that we are interested in; capturing at least certain environmental features when modelling computer-based systems is essential. Moreover, it is crucial to understand and capture human behaviour and their interactions with the system in order to be able to assess or even predict possible failures.

One approach that considers the human in systems where human-computer interactions are highly critical is the work by J. Rushby as described in [79], among others. To try to analyse how errors can result from human-computer interaction, the approach compares what is called the *mental* model of an operator (system user) and the *system* model. The mental model corresponds to the model the operator believes to be the real model of the system. Both the system model and the mental model are described as finite state transition systems and checked for consistency using a mechanised formal method. The outcome of such a check suggests places where design should be improved.

In any case, to model human behaviour there is a need to borrow concepts and models from other disciplines like Cognitive Science and/or Artificial Intelligence. A combined formal approach can then be used to describe software systems and part of their environment, as well as their interactions. Verification tools based on such combined formalisms would make it possible to verify for instance dependable systems with human-computer interaction.

The design of computer-based systems like ParcelCall requires expertise from different disciplines. Human factors and organisational knowledge are important skills to define the systems requirements and how to deploy the system into a particular context. Engineering skills become crucial in designing, developing and testing. Most of the expected outcomes overlap different kinds of expertise. Hence a production environment requires all these different kinds of

expertise and productive cooperation. This is a major issue in organising and managing multi-disciplinary development environments. Solving some issues in the presence of multi-disciplinarity can imply to turn into inter-disciplinarity, that is, the development environment has evolved in such a way to create its own inter-disciplinary settings. The mechanisms behind this evolution from multi-disciplinarity to inter-disciplinarity are still vaguely understood. Moreover, how to translate the above concepts into design, i.e., how to move from a multi-disciplinary to an inter-disciplinary design of computer-based systems, is challenging for future research.

5.2.2 Design for Evolution

Evolution is one of the most critical issues in the design of computer-based systems. Systems are already obsolete when they are delivered either because the environment has changed or stakeholders have changed their understanding about the system and its requirements. One of the reasons for this is that systems are engineered following a *static* paradigm. Even iterative development processes like the spiral model only capture evolution to a limited extent.

Experience shows [1, 2, 4, 68, 90] that it is not possible to freeze requirements at any stage of the life cycle. Designing phases are organised in terms of beginnings, ends, deadlines, that often developing environments fail to meet. Systems evolution is mainly considered within maintenance. But the evolution of computer-based systems is a concept broader than maintenance. Evolution actually starts as soon as a business case identifies a particular system, that is, evolution starts even before the system exists. Unfortunately most of the methodologies provide little support to evolution. Hence the need to develop new methodologies supporting system evolution. We started in different contexts to analyse requirements evolution [1, 2, 4]. The analysis of the taxonomy of requirements changes and product features has improved our understanding of the specific case study. Our previous experience should be taken into consideration while designing ParcelCall. The design context should register symptoms of evolution, otherwise it will not be possible to understand evolutionary information. Evolution stresses a different way of interpreting system design. Future research should investigate how methodologies can support *design for evolution.* In this new perspective, design and evolution are at the same level. Systems do not evolve, if we do not design their evolution.

5.3 A Research Agenda

The previous section points out some limits related to modelling and evolving ParcelCall. As more changes to the system are necessary, the complexity of the system increases. The management of the system's complexity without any methodological support for understanding and bounding the side effects all over the life cycle will collapse triggering subsequent complexity explosions. There are still few methods to analyse and design scalable complexity. Our analysis of

the ParcelCall case study suggests some limits in modelling such systems due to their inner complexity. The modelling issues and the evolutionary perspectives point out the crucial importance of analysing, modelling and evolving computer-based systems. It emphasises how development depends on these three aspects and how which they be understood as orthogonal to the entire life cycle. Design of computer-based systems should be based on an integration of analysis, modelling and evolution. Failing in taking into account one of these aspects will affect our ability to deploy evolvable computer-base systems.

A mature development environment should be able to analyse its own multi-disciplinarity in order to understand which inter-disciplinarity it is able to deploy. In other words, this inter-disciplinarity should match the one required by the specific computer-based system that the development environment aims to produce. To analyse multi-disciplinarity in a context and understand the process to deploy inter-disciplinarity into the development of computer-based systems is a key factor to success.

Experience shows that computer-based systems do fail. Consequently, these systems should be designed in such a way that considers faults. Evolution provides new insights in how to handle faults and should thus be considered in the design of such systems. This implies a new strategy in designing computer-based systems. The deployment of quality (dependable or survivable) computer-based systems can be obtained by designing its evolution to reach continuous quality (dependability or survivability). Quality (dependable or survivable) computer-based systems depend on our ability to react to failures by evolution.

Our analysis of ParcelCall points out research directions to improve our ability in modelling and evolving computer-based systems. In particular, in order to deal with evolution of computer-based systems we are currently devising an empirical framework to analyse rough data [4]. Industry collects a massive quantity of data, which is not analysed. This is because there is little support to analysing and structuring life cycle data. A systematic and methodological data analysis should be part of the corporate culture and not considered as an extra and expensive activity. Analysis is the only way to provide feedback to design and organisation.

The empirical framework represents the basis for understanding how requirements of computer-based systems evolve. This will allow to identify the stable and changeable parts of a computer-based system and making this information available for future developments. In contrast to the widely held perception about evolution, we consider evolution a paradigm for designing computer-based systems. Hence we intend to investigate evolutionary tools and methodologies for designing. Our analysis aims to take into account a multi-disciplinary perspective in order to link and understand socio and technical evolutions. The evolutionary perspective aims moreover to understand how evolution influences undependability and may influence dependability of computer-based systems.

ParcelCall made clear the current limits in expressing multi-perspective modelling. In order to improve our ability to model computer-based systems we need to understand how to incorporate human-factors information. Several questions can be formulated on our case study providing essential information which is,

however, not further taken into account in current modelling techniques. Such questions include: how does the localisation system influence the carrier system in terms of organisation and culture; how do people view and cope with the information provided by the localisation system; how do different groups within a carrier organisation conceptualise and represent temporal issues; what is the impact of temporal validity of information on the working procedure of the humans in the carrier system; how are planning and routing decisions changed; how is status and localisation information represented for the different parties involved in the transportation network from carriers or subcarriers to customers; how accurate should provided information be; and so on.

Furthermore, current methodologies provide little guidance (or none) in how to integrate non-functional requirements into modelling. Simulation or verification tools could provide some feedback on the satisfiability of non-functional requirements which cut across different components of the system. The development of tools requires a formal modelling approach. In [46] we provide a logical framework to formalise the pragmatic extension of UML for component-based specification from [15]. Our framework consists of a distributed temporal logic MDTL which allows us to describe for instance local and global properties of components and component interactions [45]. With our framework we can already describe interactions between the several components within the ParcelCall system; between the customers and ParcelCall (in this case the component GIS); and between the legacy system of the carrier and ParcelCall (in this case component GTS). In the last case, we can also deal with the interactions caused by human carriers and ParcelCall. We can, however, not describe the human carrier perception of ParcelCall. We are considering an extension of our distributed logic to incorporate agent logics (essentially logics of knowledge of belief) for describing relevant aspects of human behaviour. How non-functional requirements can be captured in our logic or how feasible such an approach is for verification needs to be investigated.

In conclusion this chapter focuses on multi-disciplinarity in design of computer-based systems with the aim of clarifying current gaps and problems. We outline the need to obtain an inter-disciplinary approach in design and discussed how modelling languages like UML offer a limited support in dealing with dependable computer-based systems. In order to model computer-based systems, as in our case the legacy system of transport and logistic companies and its integration with the localisation system offered by ParcelCall, an approach like UML is not satisfactory. Several non-functional requirements arising from diverse perspectives on the system including those of disciplines like cognitive science, artificial intelligence, sociology and psychology, cannot be captured. A more powerful framework emerging from combined efforts of the different disciplines is required. Furthermore, the analysis of the case study points out limits in modelling *evolving* computer-based systems. Such considerations motivate our ongoing and future research. Finally, we believe that considerations made in this paper can be beneficial to other people approaching or dealing with the design of dependable computer-based systems.

Chapter 6

Conclusions

This report provides new insights on Requirements Evolution. The empirical investigations of industrial case studies provided valuable input to enhance our understanding about Requirements Evolution. The main contribution of this work are summarised in what follows.

Taxonomy of Evolution. This report emphasises different aspects of evolution. Evolution is a broadly accepted concept, but people actually refer to different evolutionary aspects. This report contributes to clarify the different viewpoints about evolution. In practical terms a taxonomy may help to improve communication and reduce misunderstandings among people interested about evolutionary aspect of computer-based systems. A taxonomy may furthermore help to identify inconsistencies due to the different understandings about evolution. This report identifies a layered structure to analyse evolution. Each layer refers to a different type of evolution. These layers point out that each evolution can differently contribute to (un)dependability. The reviews of different evolutionary models points out our ability in representing evolutionary aspects of computer-based systems. This report points out relationships between the taxonomy of evolution and dependability. Further research should analyse these relationships.

Empirical Requirements Evolution. The empirical analyses of industrial case studies represent a valuable experience to understand the nature of Requirements Evolution. The empirical results points out product related features, which may enhance our ability in monitoring and controlling Requirements Evolution in industrial settings. This report provides new insights in measuring evolutionary aspects of requirements. This report proposes new metrics to evaluate aspects of evolutionary requirements. The new metrics have been assessed on the industrial dataset. This represents a valuable experience, which may be replicated in other industrial contexts. The empirical results points out the need to enhance our ability in capturing evolutionary aspects of requirements by quantitative approaches.

Requirements Evolution Modelling. The empirical investigation provides input to our modelling of Requirements Evolution. Requirements Evolution is structured in terms of requirements changes (i.e., added, deleted and modified requirements). This devises a work flow representation of Requirements Evolution. The work flow modelling captures requirements evolution. The graphical representation stresses product oriented features, which may enhance our understanding of evolutionary features.

Design for Evolution. This report stresses the importance of evolution for computer-based systems. Systems are already obsolete when they are delivered either because the environment has changed or stakeholders have changed their understanding about the system and its requirements. One of the reasons for this is that systems are engineered following a static paradigm. Even iterative development processes like the spiral model only capture evolution to a limited extent. Experience shows that it is not possible to freeze requirements at any stage of the life cycle. Designing phases are organised in terms of beginnings, ends, deadlines, that often developing environments fail to meet. Systems evolution is mainly considered within maintenance. But the evolution of computer-based systems is a concept broader than maintenance. Evolution actually starts as soon as a business case identifies a particular system, that is, evolution starts even before the system exists. Unfortunately most of the methodologies provide little support to evolution. Hence the need to develop new methodologies supporting system evolution. We started in different contexts to analyse requirements evolution. The analysis of the taxonomy of requirements changes and product features has improved our understanding of the specific case study. Our previous experience should be taken into consideration while designing ParcelCall. The design context should register symptoms of evolution, otherwise it will not be possible to understand evolutionary information. Evolution stresses a different way of interpreting system design. Future research should investigate how methodologies can support design for evolution. In this new perspective, design and evolution are at the same level. Systems do not evolve, if we do not design their evolution.

In conclusions this report provides a detailed account of requirements evolution in industrial contexts. The empirical results provide new insights enhancing our understanding of requirements evolution. The empirical nature of this work supports the validity of the evolutionary results. This report and the discussed results may be beneficial for practical requirements evolution.

Bibliography

[1] Stuart Anderson and Massimo Felici. Controlling requirements evolution: An avionics case study. In *Proceedings of SAFECOMP 2000, 19th International Conference on Computer Safety, Reliability and Security*, LNCS 1943, pages 361–370, Rotterdam, The Netherlands, October 2000. Springer-Verlag.

[2] Stuart Anderson and Massimo Felici. Requirements changes risk/cost analyses: An avionics case study. In M.P. Cottam, D.W. Harvey, R.P. Pape, and J. Tait, editors, *Foresight and Precaution, Proceedings of ESREL 2000, SARS and SRA-EUROPE Annual Conference*, volume 2, pages 921–925, Edinburgh, Scotland, United Kingdom, May 2000.

[3] Stuart Anderson and Massimo Felici. Requirements engineering questionnaire, version 1.0, January 2001.

[4] Stuart Anderson and Massimo Felici. Requirements evolution: From process to product oriented management. In *Proceedings of Profes 2001, 3rd International Conference on Product Focused Software Process Improvement*, LNCS 2188, pages 27–41, Kaiserslautern, Germany, September 2001. Springer-Verlag.

[5] Lowell Jay Arthur. *Rapid Evolutionary Development: Requirements, Prototyping & Software Creation*. John Wiley & Sons, 1992.

[6] K. Suzanne Barber et al. Requirements evolution and reuse using the systems engineering process activities (sepa). *Australian Journal of Information Systems (AJIS)*, 7:75–97, 1999. Special Issue on Requirements Engineering.

[7] Daniel M. Berry and Brian Lawrence. Requirements engineering. *IEEE Software*, pages 26–29, March 1998.

[8] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[9] Barry W. Boehm. Software engineering economics. *IEEE Transaction on Software Engineering*, 10(1):4–21, January 1984.

[10] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(2):61–72, May 1998.

[11] Barry W. Boehm et al. *Software Cost Estimation with COCOMO II*. Prentice-Hall, 2000.

[12] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[13] David Bustard, Peter Kawalek, and Mark Norris, editors. *Systems Modeling for Business Process Improvement*. Artech House, 2000.

[14] CCTA, editor. *Prince2 Manual - Managing successful projects with PRINCE 2*. CCTA - Central Computer and Telecommunications Agency, 1998.

[15] J. Cheesman and J. Daniels. UML Components. Component Software Series. Addison-Wesley, 2001.

[16] Elayne Coakes, Dianne Willis, and Raymond Lloyd-Jones, editors. *The New SocioTech: Graffiti on the Long Wall*. CSCW. Springer, 2000.

[17] CREWS Project. Cooperative Requirements Engineering With Scenarios. http://sunsite.informatik.rwth-aachen.de/CREWS/.

[18] CSEG. Cooperative Systems Engineering Group, Computing Department, Lancaster University, http://www.comp.lancs.ac.uk/computing/research/cseg/.

[19] Alan M. Davis and Pei Hsia. Giving voice to requirements engineering. *IEEE Software*, pages 12–16, March 1994.

[20] Giorgio De Michelis et al. A three-faceted view of information systems. *Communications of the ACM*, 41(12):64–70, December 1998.

[21] Ralf Dömges and Klaus Pohl. Adapting traceability environments to project-specific needs. *Communications of the ACM*, 41(12):54–62, December 1998.

[22] E. Edwards. Man and machine: Systems for safety. In *Proceedings of British Airline Pilots Associations Technical Symposium*, pages 21–36, London, 1972. British Airline Pilots Associations.

[23] Massimo Felici and Juliana Küster Filipe. Limits in modelling evolving computer-based systems. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2002*, Madrid, Spain, March 2002.

[24] Massimo Felici, Alberto Pasquini, and Mark-Alexander Sujan. Applicability limits of software reliability growth models. In *MMR'2000, Deuxième Conférence Internationale sur les Méthodes Mathématiques en Fiabilité: Méthodologie, Pratique et Inférence*, volume 1, pages 397–400, Bordeaux, France, July 2000.

[25] Massimo Felici, Mark-Alexander Sujan, and Maria Wimmer. Integration of functional congnitive and quality requirements: A railways case study. In Katrina Maxwell, Rob Kusters, Erik van Veenendaal, and Adrian Cowderoy, editors, *Project Control: The Human Factor. Proceedings of ESCOM-SCOPE 2000*, pages 395–403, Munich, Germany, April 2000.

[26] Massimo Felici, Mark-Alexander Sujan, and Maria Wimmer. Integration of functional congnitive and quality requirements. A railways case study. *Information and Software Technology*, 42(14):993–1000, November 2000.

[27] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, second edition, 1996.

[28] James Fleck. Learning by trying: the implementation of configurational technology. *Research Policy*, 23:637–652, 1994.

[29] Brian Foote and Joseph Yoder. Evolution, architecture, and metamorphosis. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*, chapter 13. Addison-Wesley, 1996.

[30] Aditya K. Ghose. Managing requirements evolution: Formal support for functional and non-functional requirements. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 77–84, Fukuoka, Japan, 1999.

[31] Tom Gilb. *Principles of Software Engineering Management*. Addison-Wesley, 1988.

[32] Theodore F. Hammer, Leonore L. Huffman, and Linda H. Rosenberg. Doing requirements right the first time. *CROSSTALK The Journal of Defense Software Engineering*, pages 20–25, December 1998.

[33] S.D.P. Harker, K.D. Eason, and J.E. Dobson. The change and evolution of requirements as a challenge to the practice of software engineering. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 266–272, San Diego, California, USA, January 1993. IEEE Computer Society Press.

[34] Herbert William Heinrich. *Industrial accident prevention: a scientific approach*. McGraw-Hill, 3rd edition, 1950.

[35] Hubert F. Hofmann and Franz Lehner. Requirements engineering as a success factor in software projects. *IEEE Software*, pages 58–66, July/August 2001.

[36] Ivy F. Hooks and Kristin A. Farry. *Customer-Centered Products: Creating Successful Products Through Smart Requirements Management*. Amacom, 2001.

[37] IEEE. *IEEE Std 982.1 - IEEE Standard Dictionary of Measures to Produce Reliable Software*, 1988.

[38] IEEE. *IEEE Std 982.2 - IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, 1988.

[39] IEEE. *IEEE Std 830 - IEEE Recommended Practice for Software Requirements Specifications*, 1993.

[40] ISO/IEC. *ISO/IEC 9126 - Information Technology - Software quality characteristics and metrics*.

[41] Matthias Jarke. Requirements tracing. *Communications of the ACM*, 41(12):32–36, December 1998.

[42] Matthias Jarke et al. Theories underlying requirements engineering: An overview of nature at genesis. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 19–31, San Diego, California, USA, January 1993. IEEE Computer Society Press.

[43] Chris F. Kemerer and Sandra Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, July/August 1999.

[44] Gerald Kotonya and Ian Sommerville. Requirements engineering with viewpoints. *Software Engineering Journal*, 11:5–18, January 1996.

[45] Juliana Küster Filipe. Fundamentals of a module logic for distributed object systems. *Journal of Functional and Logic Programming*, 3, March 2000.

[46] Juliana Küster Filipe. A logic-based formalization for component specification. In *To appear*, 2002.

[47] Juha Kuusela. Architectural evolution. In Patrick Donohoe, editor, *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 471–478, San Antonio, Texas, USA, 1999. IFIP, Kluwer Academic Publishers.

[48] W. Lam. Achieving requirements reuse: A domain-specific approach from avionics. *The Journal of Systems and Software*, 38(3):197–209, September 1997.

[49] W. Lam, J.A. McDermid, and A.J. Vickers. Ten steps towards systematic requirements reuse. In *Proceedings of the Third IEEE International Symposium on Requirements Engineering*, pages 6–15, Annapolis, Maryland, USA, January 1997. IEEE Computer Society Press.

[50] Jean-Claude Laprie. Dependable computing: Concepts, limits, challenges. In *FTCS-25, the 25th IEEE International Symposium on Fault-Tolerant Computing - Special Issue*, pages 42–54, Pasadena, California, USA, June 1995.

[51] Jean-Claude Laprie et al. Dependability handbook. Technical Report LAAS Report no 98-346, LIS LAAS-CNRS, August 1998.

[52] Mingjune Lee and Barry W. Boehm. The winwin requirements negotiation system: A model-driven approach. Technical Report USC-CSE-96-502, University of Southern California, 1996.

[53] M.M. Lehman. Software's future: Managing evolution. *IEEE Software*, pages 40–44, Jan-Feb 1998.

[54] M.M. Lehman and L.A. Belady. *Program Evolution: Processes of Software Change*, volume 27 of *A.P.I.C. Studies in Data Processing*. Academic Press, 1985.

[55] M.M. Lehman, D.E. Perry, and J.F. Ramil. On evidence supporting the feast hypothesis and the laws of software evolution. In *Proceedings of Metrics '98*, Bethesda, Maryland, November 1998.

[56] Nancy G. Leveson. *SAFEWARE: System Safety and Computers*. Addison-Wesley, 1995.

[57] Frederick Levine, Christopher Locke, David Searls, and David Weinberger. *The Cluetrain Manifesto: The end of business as usual*. ft.com, 2000.

[58] Bev Littlewood, Peter Popov, and Lorenzo Strigini. Modelling software design diversity: a review. *ACM Computing Surveys*, 33(2):177–208, 2001.

[59] Bev Littlewood and Lorenzo Strigini. Software reliability and dependability: a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 177–188. ACM Press, Limerick, June 2000.

[60] Michael R. Lyu, editor. *Handbook of Sofwtare Reliability Engineering*. IEEE Computer Society Press, 1996.

[61] Saeko Matsuura, Hironobu Kuruma, and Shinichi Honiden. Eva: A flexible programming method for evolving systems. *IEEE Transactions on Software Engineering*, 23(5):296–313, May 1997.

[62] NATURE Project. Novel Approaches to Theories Underlying Requirements Engineering. http://www-i5.informatik.rwth-aachen.de/PROJEKTE/NATURE/nature.html.

[63] Donal A. Norman. *The Invisible Computer*. The MIT Press Cambridge, Massachusetts, 1998.

[64] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, October 1994.

[65] OMG Unified Modeling Language Revision Task Force. OMG Unified Modeling Language Specification, version 1.4 draft edition, February 2001.

[66] Charles Perrow. *Normal Accidents: Living with High-Risk Technologies*. Princeton University Press, 1999.

[67] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice-Hall, 1998.

[68] PROTEUS Project. Meeting the challenge of changing requirements. Deliverable 1.3, Centre for Software Reliability, University of Newcastle upon Tyne, June 1996.

[69] Balasubramaniam Ramesh. Factors influencing requirements traceability practice. *Communications of the ACM*, 41(12):37–44, December 1998.

[70] Brian Randel. Facing up to faults. *Computer Journal*, 43(2):95–106, 2000.

[71] James Reason. *Managing the Risks of Organizational Accidents*. Ashgate Publishing Limited, 1997.

[72] RENOIR Project. Requirements Engineering Network Of International cooperating Research groups a network of excellence. http://www.cs.ucl.ac.uk/research/renoir/.

[73] REVERE Project. REVerse Engineering of REquirements to support business process change, Computing Department, Lancaster University, UK. http://www.comp.lancs.ac.uk/computing/research/cseg/projects/revere/.

[74] RISE. Research Institute in Software Evolution, Computer Science, University of Durham, UK. http://www.dur.ac.uk/CSM/.

[75] James Robertson and Suzanne Robertson. Volere: Requirements specification template. Technical Report Edition 6.1, Atlantic Systems Guild, 2000.

[76] Suzanne Robertson and James Robertson. *Mastering the Requirements Process*. Addison-Wesley, 1999.

[77] RTCA. *DO-178B Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[78] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[79] John Rushby. Modeling the human in human factors: Extended abstract. In *Proceedings of Safecomp 2001, The 20th International Conference on Computer, Safety and Reliability*, LNCS 2187, pages 86–91, Budapest, Hungary, September 2001. Springer-Verlag.

[80] W. J. Salamon and D. R. Wallace. Quality characteristics and metrics for reusable software. Technical Report NISTIR 5459, NIST, 1994.

[81] Pete Sawyer, Ian Sommerville, and Stephen Viller. Capturing the benefits of requirements engineering. *IEEE Software*, pages 78–85, March/April 1999.

[82] Norman F. Schneidewind. Measuring and evaluating maintenance process using reliability, risk, and test metrics. *IEEE Transactions on Software Engineering*, 25(6):769–781, November/December 1999.

[83] Lui Sha, Ragunathan Rajkumar, and Michael Gagliardi. A software architecture for dependable and evolvable industrial computing systems. Technical Report CMU/SEI-95-TR-005, CMU/SEI, July 1995.

[84] J. Siddiqi and M.C. Shekaran. Requirements engineering: The emerging wisdom. *IEEE Software*, pages 15–19, March 1996.

[85] Ian Sommerville. *Software Engineering*. Addison-Wesley, sixth edition, 2000.

[86] Ian Sommerville, Gerald Kotonya, Steve Viller, and Pete Sawyer. Process viewpoints. Technical Report CSEG/1/1995, Lancaster University, 1995.

[87] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, 1997.

[88] Ian Sommerville and Pete Sawyer. Viewpoints: principles, problems and a practical approach to requirments engineering. *Annals of Software Engineering*, pages 101–130, 1997.

[89] Ian Sommerville, Pete Sawyer, and Stephen Viller. Viewpoints for requirements elicitation: a practical approach. In *Proceedings of the IEEE International Conference on Requirements Engineering*, Colorado Springs, Colorado, April 1998.

[90] George Stark, Al Skillicorn, and Ryan Ameele. An examination of the effects of requirements changes on software releases. *CROSSTALK The Journal of Defence Software Engineering*, pages 11–16, December 1998.

[91] Neil Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[92] Stephen Viller and Ian Sommerville. Social analysis in the requirements engineering process: from ethnography to method. Technical Report CSEG/14/1998, Lancaster University, 1998.

[93] Gerald M. Weinberg. *Quality Software Management. Volume 4: Anticipating Change*. Dorset House, 1997.

[94] David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[95] Karl Eugene Wiegers. *Software Requirements*. Microsoft Press, 1999.

[96] Robin Williams, Roger Slack, and James Stewart. Social learning in multimedia. Final report, EC targeted socio-economic research, project: 4141 PL 951003, Research Centre for Social Sciences, The University of Edinburgh, January 2000.

[97] D. Zowghi, A. K. Ghose, and P. Peppas. A framework for reasoning about requirements evolution. In *Proceedings of PRICAI '96*, number 1114 in LNAI, pages 157–168. Springer-Verlag, 1996.

[98] Didar Zowghi and Ray Offen. A logical framework for modeling and reasoning about the evolution of requirements. In *Proceedings of the Third IEEE International Symposium on Requirements Engineering*, pages 247–257, Annapolis, Maryland, USA, January 1997. IEEE Computer Society Press.