Observational Models of Requirements Evolution

Massimo Felici

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2004

Abstract

Requirements Evolution is one of the main issues that affect development activities as well as system features (e.g., system dependability). Although researchers and practitioners recognise the importance of requirements evolution, research results and experience are still patchy. This points out a lack of methodologies that address requirements evolution. This thesis investigates the current understanding of requirements evolution and explores new directions in requirements evolution research. The empirical analysis of industrial case studies highlights software requirements evolution as an important issue. Unfortunately, traditional requirements engineering methodologies provide limited support to capture requirements evolution. Heterogeneous engineering provides a comprehensive account of system requirements. Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings. The formal extension of a heterogeneous account of requirements provides a framework to model and capture requirements evolution. The application of the proposed framework provides further evidence that it is possible to capture and model evolutionary information about requirements. The discussion of scenarios of use stresses practical necessities for methodologies addressing requirements evolution. Finally, the identification of a broad spectrum of evolutions in socio-technical systems points out strong contingencies between system evolution and dependability. This thesis argues that the better our understanding of socio-technical evolution, the better system dependability. In summary, this thesis is concerned with software requirements evolution in industrial settings. This thesis develops methodologies to empirically investigate and model requirements evolution, hence *Observational Models of Requirements Evolution*. The results provide new insights in requirements engineering and identify the foundations for Requirements Evolution Engineering. This thesis addresses the problem of empirically understanding and modelling requirements evolution.

Acknowledgements

I would like to thank my supervisors, Stuart Anderson and Perdita Stevens. Special thanks go to Stuart Anderson, my principal supervisor, who has challenged me throughout my studies by supporting my research interest and vision on a novel and risky subject. Tracking back where all this started, I have to acknowledge the EU OLOS network for supporting my visit to Edinburgh prior to these studies. The persons responsible for that Scottish visit were Alberto Pasquini, my former supervisor at the Italian National Agency for New Technologies, Energy and Environment (ENEA), who sent me to Edinburgh and, again, Stuart Anderson, who hosted me in Edinburgh since then.

Finally, but not the least, I would like to thank my parents, Gaetano Felici and Adelina Caruso, and my brother Marco Felici for their support throughout my studies. I would like also to thank Friends, Colleagues and Others who were unnoticeably supportive. Thanks to you all.

This thesis, like any project, would have been impossible without any financial support. I am grateful to the following grants and organisations:

- The Laboratory for Foundations of Computer Science, LFCS, at the University of Edinburgh for funding the fees of the first year of my studies
- The University of Catania, Italy, for supporting the first year of my studies by a one-year grant for attending abroad post-graduate courses in foreign universities
- The Interdisciplinary Research Collaboration in Dependability of Computer-based Systems, DIRC, UK EPSRC grant GR/N13999, for supporting my work on Requirements Evolution for Design for Dependability
- The Italian National Research Council, CNR, for supporting my research project "Requirements Evolution: Understanding Formally Software Engineering Processes within Industrial Contexts", Bando n. 203.15.11
- The Italian National Research Council, CNR, for supporting my research project "A Formal Framework for Requirements Evolution", Bando n. 203.01.72, Codice n. 03.01.04.

The following publications contain early parts of this thesis:

[Anderson and Felici, 2000a] Stuart Anderson and Massimo Felici. Controlling Requirements Evolution: An Avionics Case Study. In Koornneef, F. and van der Meulen, M., editors, Proceedings of the 19th International Conference on Computer Safety, Reliability and Security, SAFECOMP 2000, LNCS 1943, pages 361-370, Rotterdam, The Netherlands. Springer-Verlag.

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

[Anderson and Felici, 2001] Stuart Anderson and Massimo Felici. Requirements Evolution: From Process to Product Oriented Management. In Bomarius, F. and Komi-Sirviö, S., editors, Proceedings of the Third International Conference on Product Focused Software Process Improvement, PROFES 2001, LNCS 2188, pages 27-41, Kaiserslautern, Germany. Springer-Verlag.

[Anderson and Felici, 2002] Stuart Anderson and Massimo Felici. Quantitative Aspects of Requirements Evolution. In Proceedings of the Twenty-Sixth Annual International Computer Software and Applications Conference, COMPSAC 2002, pages 27-32, Oxford, England. IEEE Computer Society.

[Felici, 2000] Massimo Felici. Dependability Perspectives in Requirements Engineering. In Student Forum, Workshops and Abstracts Proccedings of The International Conference on Dependable Systems and Networks, DSN 2000, pages A43-A45, New York, New York, USA. IEEE Computer Society.

[Felici, 2003] Massimo Felici. Taxonomy of Evolution and Dependability. In Proceedings of the Second International Workshop on Unanticipated Software Evolution, USE 2003, pages 95-104, Warsaw, Poland.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Massimo Felici*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Requirements Evolution is an emerging phenomenon of any software related project. The cost and risk associated with requirements changes inevitably increase with the progress of software projects [Boehm, 1981, Boehm, 1984]. Requirements change can prevent projects from ending successfully. They can also affect the main system functionalities by giving rise to uncertainties of critical system features (e.g., dependability, safety, reliability, security, etc.). These issues motivate the steadily growing interest in requirements engineering[1].

Any software production involves diverse stakeholders, who interact each other by means of development deliverables (e.g., requirements specification, system design, software code, etc.), processes (e.g., change management, software development, etc.) and activities (e.g., requirements elicitation, software testing, etc.). Effective cooperation needs stakeholders to understand their different viewpoints on software projects [Sommerville and Sawyer, 1997a]. On one hand viewpoints identify different system perspectives (usually associated with different stakeholders). On the other hand viewpoints support the focused analysis of system requirements. Unfortunately, poor coordination and understanding of viewpoints inhibit the elicitation of requirements and affect requirements consistency [Jirotka and Goguen, 1994, Sommerville and Sawyer, 1997a]. Software is therefore the result of engineering tech-

[1]Various books, conferences and journals, e.g., [Davis and Hsia, 1994, Siddiqi and Shekaran, 1996, Berry and Lawrence, 1998], give an account of the debates on research and practice in requirements engineering.

nical solutions through stakeholder interactions[2]. These interactions influence how stakeholders acquire their knowledge as well as system design [Bijker et al., 1989, Jirotka and Goguen, 1994, Vincenti, 1990]. The way interactions capture and shape stakeholder knowledge and system design manifests over software projects as requirements changes, hence requirements evolution. Stakeholder interactions, cooperations and negotiations result in shifts in the grounds for agreement. These shifts drive requirements evolution. The problem is how to model requirements evolution in order to capture stakeholder interactions through requirements. Although understanding stakeholder interactions highlights requirements evolution, poor understanding of the mechanisms of requirements evolution affects stakeholder interactions. This often results in poor requirements baselines that affect software production as well as system features.

1.1 Thesis Contribution

This thesis considers requirements evolution as an unavoidable feature of software production. Classically, requirements evolution is seen as an error in the engineering process. In contrast, this thesis takes into account requirements evolution as an essential feature of good design processes [Petroski, 1992, Petroski, 1994]. Diverse production aspects may trigger requirements evolution, which propagates through requirements changes that affect development activities and processes. In any development process the definition of requirements is the first phase and it is always crucial for the success of software projects.

This thesis considers software requirements evolution within industrial production environments. In contrast to the process-centred approach taken in current requirements engineering practice, this thesis takes a product-centred approach based on empirical analysis and modelling. 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. These case studies provide a detailed account of the cooperative processes adopted by stakeholders. The underlying hypoth-

[2]The mechanisms underlying the social design and implementation of technology systems are referred to as the *Social Shaping of Technology* (SST) [Williams and Edge, 1996, MacKenzie and Wajcman, 1999].

esis of this thesis is that stakeholder interaction in cooperative processes is a powerful driver of requirements evolution. This thesis addresses the lack of understanding about requirements evolution. It enhances our ability to understand and model requirements evolution.

1.2 Thesis Synopsis

This thesis is structured as follows. Chapter 2 reviews related work in requirements engineering. Although requirements evolution is a recognised phenomenon of software systems, requirements engineering research and practice mainly focus on management aspects. Management methodologies advocate process-oriented approaches in order to tackle requirements changes. On one hand these methodologies allow standardisation and organise work practice, although they provide limited support to tailor processes in order to capture system features. On the other hand system features pervade processes as well as organisations. Requirements evolution therefore becomes an emergent phenomenon as well as an intrinsic feature of software systems. The review points out a rationale for a framework that supports the analysis, control and monitor of requirements evolution, hence requirements evolution. Related research and the existence of few empirical studies motivated the investigation of case studies drawn from industry.

Chapter 3 contains an empirical investigation of an avionics safety-critical case study drawn from industry with respect to requirements evolution. The investigation of an industrial case study allows us to acquire input from practice in requirements engineering. This is to take a realistic account of requirements evolution. Moreover, a case study provides domain knowledge that characterises the specific industrial context. Case studies drawn from industry stress the crucial aspect of domain knowledge. Most of requirements engineering methodologies have serious practical limitations, because they lack of domain knowledge. The empirical analysis points out evolutionary aspects of requirements as well as practical issues (e.g., poor support for the analysis of requirements evolution in live production environments). The case study furthermore stresses the need to enhance our understanding of requirements evolution.

Chapter 4 describes a case study drawn from smart card industry. Despite less data than the avionics case study, the analysis of requirements viewpoints allows us further to understand how different system perspectives may result in different requirements changes. The analysis relies on interviews and questionnaires. In spite of sparse data, the analysis points out many issues that characterise live production environments. Although changes affect several viewpoints and increase project risk, they are part of learning and understanding processes in software production. The analysis highlights how even a single change affects many different socio-technical aspects.

Chapter 5 introduces a formal framework to model and capture requirements evolution. The framework relies on a heterogeneous account of requirements. Heterogeneous engineering provides a comprehensive account of system requirements. Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings. The formal extension of solution space transformation defines a framework to model and capture requirements evolution. The resulting framework is sufficient to interpret requirements changes, hence requirements evolution. The formal framework captures how requirements evolve through subsequent releases. Hence, it is possible to define requirements evolution in terms of sequential solution space transformations. Intuitively, requirements evolution identifies a path that browses solution spaces.

Requirements management methodologies and practices rely on requirements traceability. Although requirements traceability provides useful information about requirements, traceability manifests emergent evolutionary aspects just as requirements do. It is also important to understand requirements dependencies that constrain software production. Requirements dependencies, as an instance of traceability, identify relationships between requirements. Moreover, requirements dependencies constrain requirements evolution. Thus, it is important to capture these dependencies in order further to understand requirements evolution. Chapter 6 shows how the formally augmented solution space transformation captures evolutionary requirements dependencies. Examples drawn from the avionics case study provide a realistic instance of requirements dependencies. These examples show how the heterogeneous framework captures evolutionary features of requirements, hence requirements evolution.

Empirical analysis and requirements evolution modelling capture evolutionary aspects of system production. Chapter 7 develops three main scenarios of practice: *Modelling Requirements Evolution*, *Process Calibration*, *Requirements Evolution Regression*. Moreover, it describes how heterogeneous requirements evolution supports the refinement of design models. Although these scenarios are descriptive, they provide an overall understanding how modelling requirements evolution enhances system production. The scenarios therefore further develop a rationale for *Requirements Evolution Engineering* (REE).

Requirements, however, represent only one aspect of socio-technical evolution. Although evolution is a necessary feature of socio-technical systems, it often increases the risk of failures. Chapter 8 reviews a taxonomy of evolution, as a conceptual framework for the analysis of socio-technical system evolution with respect to dependability. The identification of a broad spectrum of evolutions in socio-technical systems points out strong contingencies between system evolution and dependability. This thesis argues that the better our understanding of socio-technical evolution, the better system dependability. Finally, Chapter 9 draws the conclusions of this thesis.

Chapter 2

Related Work

This chapter reviews research and practice relevant to *requirements evolution*. Despite requirements evolution being a recognised phenomenon of software systems, requirements engineering research and practice mainly focuses on management aspects. Management methodologies advocate process-oriented approaches in order to tackle requirements changes. On one hand these methodologies allow standardisation and organise work practice, although they provide limited support to tailor processes in order to capture system features. On the other hand system features pervade processes as well as organisations. This chapter constructs a rationale for requirements evolution. Requirements evolution therefore becomes an emergent phenomenon as well as an intrinsic feature of software systems.

2.1 On Requirements Engineering

Requirements engineering has emerged as a novel discipline within *software engineering*. Like most research that is close to exploitation in practice, economic factors have driven the development of requirements engineering. Early research [Boehm, 1981, Boehm, 1984] in software engineering shows that it is economically convenient and effective to fix faults as early as possible in the development process. Jointly, the awareness that most software projects fail due to customer dissatisfaction triggered an increasing interest in system requirements. Since the early stages of the development of requirements engineering researchers and practitioners have been engaged in discussions about what system requirements are. The main propaganda is that requirements are concerned with *"what the system should do"*, whereas design is concerned with *"how the system should do it"*. In practice the distinction between "what" and "how" is often unclear [Sommerville and Sawyer, 1997a]. Moreover, experience points out that misunderstandings and disagreements about "what" and "how" are just fertile ground for system failures [Leveson, 1995, Perrow, 1999, Storey, 1996].

System failures and flawed designs have been often traced back to poor requirements. The diverse concerns about requirements led to the development of various methodologies, which tackle the requirements problem at different stages in the system production. The goal of *fixing* requirements has driven most early research effort [Weinberg, 1997]. Many methodologies aim to enhance requirements *correctness* and *completeness*. On the other hand many others stress the importance of effective development processes, like in the manufacturing industry. The deceptive similarities between software production and manufacturing stimulated the blossoming of many software development processes. All of them give great credit to the organisation of software processes in terms of development phases and activities. The process complexity reflects to some extent the account given to each development phase. The understandings of requirements and requirements processes differ in each development process. Although, regardless product features, they advocate that quality processes produce quality products.

Requirements engineering nowadays comprises a broad range of diverse disciplines. Most recent methodologies take into account *human factors* in software systems. On one hand requirements should also capture user needs. On the other hand requirements are a means of communication between stakeholders (e.g., project managers, software engineers, system users, regulators, etc.). Thus, requirements have a pivotal role for stakeholder interactions. Requirements need to integrate and embrace different viewpoints, hence *requirements viewpoints* [Kotonya and Sommerville, 1996, Sommerville and Sawyer, 1997b]. Multidisciplinary aspects therefore pervade and surround requirements. Studies on socio-technical systems [Coakes et al., 2000] further recognise this multidisciplinarity. The foundations of system design are grounded in heterogeneous engineering [Bijker et al., 1989]. Requirements engineering captures the social shaping [MacKenzie and Wajcman, 1999] of software systems.

The remainder of this section reviews relevant literature in requirements engineering. The review stresses those aspect mostly relevant to requirements evolution: *software and requirements processes*, *empirical requirements engineering*, *modelling in requirements engineering* and *modelling requirements evolution*.

2.1.1 Software and Requirements Processes

Software production relies on different processes defined in terms of phases and activities. Each phase and activity contribute towards different project deliverables (e.g., system design, software code, etc.). Among the different deliverables requirements have distinguished themselves within any software production. It is actually possible to identify two main distinct processes: the *software process* and the *requirements process*, also called the *Twin Processes* [Weinberg, 1997]. The software process is to deliver software systems. Whereas, the requirements process is to deliver software requirements. Thus, the two processes are mutually controlling twin processes. In mutually controlling processes, the requirement process is to narrow the discrepancy between what is wanted and what is documented. On the other hand discrepancies give information needed for control. They may differently run in the life cycle, but they are always present, exercising feedback control over the other. Empirical evidence should mutually control each process. Empirical evidence, drawn from mutually controlling the two processes, furthermore provides a basis for changing requirements processes [Weinberg, 1997]. Each Development context differently and diversely interprets relationships between software processes and requirements processes. Although it is possible to identify these two processes, the requirement process is often a phase or activity of the software development process. As in manufacturing, software and requirements engineering acknowledge various processes, usually, defined in terms of development phases and activities. Each development process[1] differently takes into account the relationships among phases and activities. In the effort to standardise soft-

[1] Any textbook in software engineering (e.g., [Pfleeger, 1998, Sommerville, 2001]) extensively introduces the most popular software development processes.

ware development few processes rose to popularity.

Figure 2.1 shows, one of the first software development processes, the *Waterfall Model*. It consists of five main phases: requirements, design, coding and unit testing, system integration, Operation and maintenance. The waterfall model implies that development phases follow each other sequentially. Thus, according to the waterfall model, the software development starts with the requirements phase and finishes with the operation and maintenance phase. The underlying hypothesis is that it is possible to discover (and freeze) all software requirements at the beginning of each project. Unfortunately, this hypothesis is unrealistic [Weinberg, 1997].

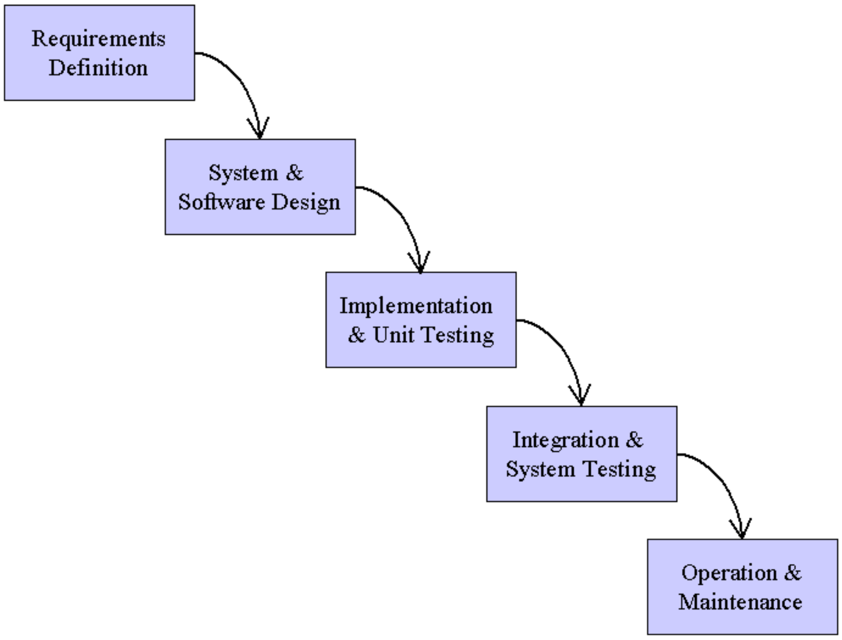


Figure 2.1: The Waterfall Model.

Many software development processes arose in order to fix the frozen-requirement assumption. The *iterated Waterfall Model* was an attempt to improve the linear cascade waterfall model. The iterated waterfall model repeats each phase, if faults are discovered in successive phases. For instance, the requirements phase is repeated every time faults are discovered during the design phase. This creates a two-flow chain development process, which traces faults back in order to fix them. Although this model is an improvement, it is still inflexible and the cost of accommodating changes is high. Figure 2.2 shows the Boehm's *Spiral Model* [Boehm, 1998], which merges risk management with software development. The spiral model consists of successive risk-driven iterations of incremental phases. The underlying hypothesis is that risk considerations can drive software production. Although, this is generally true, risk considerations drive the spiral model to behave like other models (e.g., waterfall, evolutionary development [Arthur, 1992], etc.) in particular (unfortunate) conditions (e.g., stable requirements, volatile user requirements, high-risk stringent requirements, etc.) [Boehm, 1998]. These cases diminish the benefit of the risk-driven spiral model.

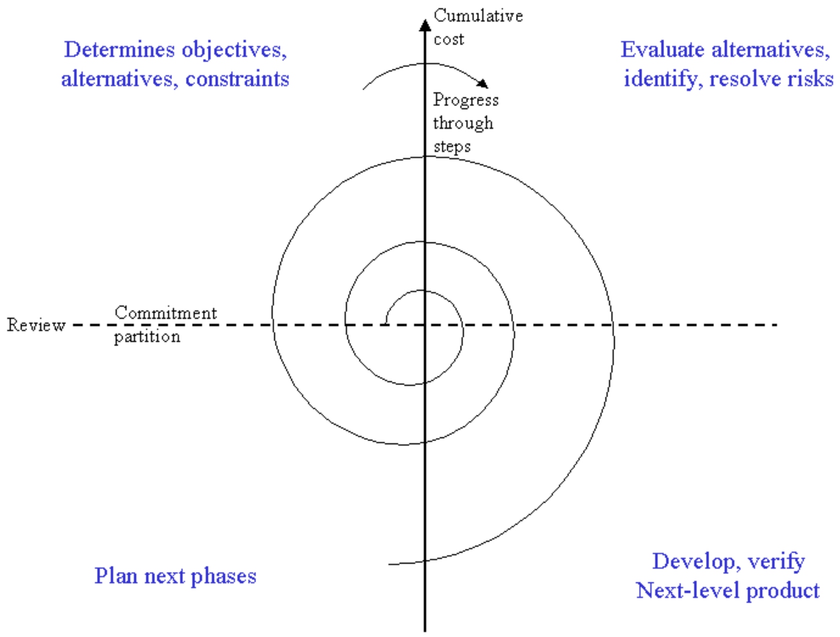


Figure 2.2: The Spiral Model.

Figure 2.3 shows another model, the *V model*. It relates testing activities with requirements and design activities. In particular, unit testing and system testing will be used to verify system design. The testing activities should guarantee that system design has been implemented correctly. Similarly, before going into operation acceptance testing should guarantee that the system complies with its high level requirements. The relationships in the V model imply that if anomalies arise during testing and verification, the requirements, design and implementation phases are repeated in order to fix any fault. The V model has been widely used in safety-critical contexts. One of the reasons is that it stresses the deliverables of each development phase [Storey, 1996]. This allows the planning of software development in terms of specific deliverables.

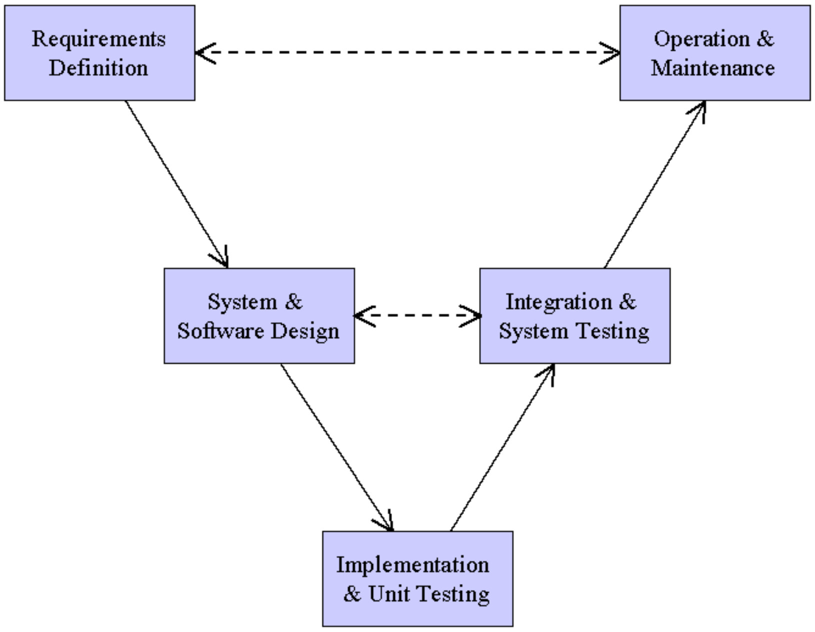
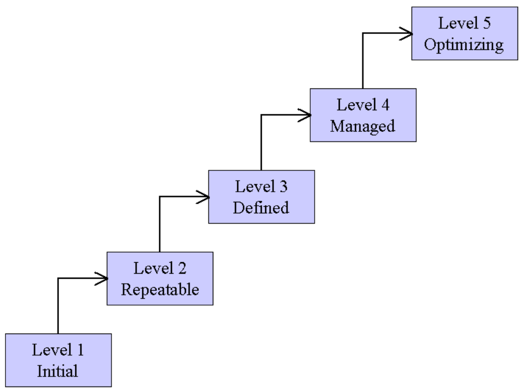


Figure 2.3: The V Model.

After the early software development processes, many other models (e.g., Unified Process [Hunt, 2000], Xtreme Programming, etc.) arose in order to adapt software production to specific design paradigms (e.g., Object-Oriented) or markets (e.g., [Levine et al., 2000]). Although they differently interpret software production, all models rely on specific development phases and activities (e.g., requirements, coding, testing, verification, etc.). They differ each other on how development phases, activities and deliverables relate each other. For instance, the *Unified Process* supports Object-Oriented design. The Unified Process is a framework which guides the tasks, people and products of the design process [Hunt, 2000]. It is a framework because it provides the inputs and the outputs of each activity, without giving any restriction how each activity must be performed. On the other hand it is called a process because its primary aim is to define: who is doing what, when they do it, how to reach a certain goal (i.e., each activity), and the inputs and outputs of each activity [Hunt, 2000]. Each phase and activity contribute towards specific deliverables (e.g., use cases, class diagrams, etc.) that capture specific information about the software system. The use cases, for instance, should capture most of the high-level system requirements. The requirements change management process is therefore tailored to the specific design methodology [Leffingwell and Widrig, 2003].

As many organisations adopted software processes, it was needed to be able to assess whether organisations were effectively adopting and improving development processes. Many standards and process improvement models (e.g., CMM, SPICE and ISO 9000) assess the extent to which organisations are able to standardise and improve their developments processes. Figure 2.4 shows the *Capability Maturity Model* (CMM) [Paulk et al., 1993]. The CMM consists of five incremental levels. Each level relies on key process areas (e.g., requirements management, software product engineering, etc.). Notice that each level introduces new key process areas and relies on the key process areas of the previous level(s). The CMM puts great emphasis on requirements, quality and change managements [Wiegers, 1999]. The incremental levels represent a progressive enhancement of software processes. Regardless the adopted software process, requirements and change management are key areas across all CMM levels.



CMM Level	Key Process Areas
Level 1	None
Level 2	Requirements management Software project planning Software project tracking and oversight Software subcontract management Software quality assurance Software configuration management
Level 3	Organization process focus Organization process definition Training program Integrated software management Software product engineering Intergroup coordination Peer reviews
Level 4	Quality process management Software quality management
Level 5	Fault prevention Technology change management Process change management

Figure 2.4: The Capability Maturity Model (CMM).

Just as new software processes arose, new requirements processes arose. Any requirement process has emerged as a very critical part affecting the whole software process. Despite their criticality, requirements processes are usually less structured than software processes. This is probably because the software and requirements processes are diverse and they deal with different problems. In general, any requirement process involves specific activities (e.g., requirements elicitation, requirements negotiation, etc.) that contribute to identify and verify software requirements. Some requirements processes may enhance particular activities (e.g., ethnographic studies, task analysis, etc.) in order to take into account various types of requirements. Sommerville and Sawyer identify guidelines of best practice in requirements engineering [Sommerville and Sawyer, 1997a]. A requirements process can therefore implement some of those guidelines. Similarly to the CMM, they also propose a requirements engineering maturity model [Sommerville and Sawyer, 1997a]. Figure 2.5 shows the requirements engineering process maturity levels.

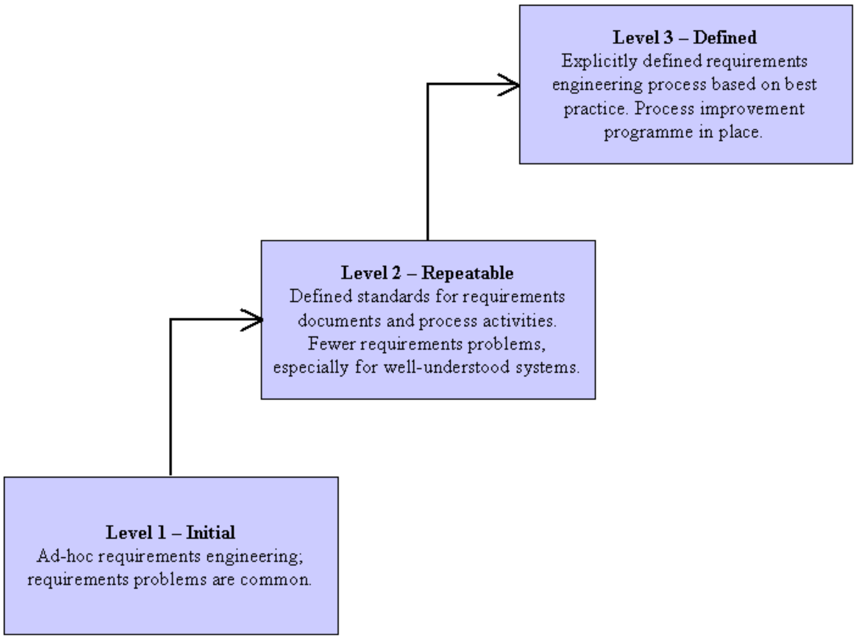


Figure 2.5: The requirements engineering process maturity levels.

The requirements engineering maturity model assesses the maturity of the requirements process adopted by an organisation. Differently from the CMM, the requirements engineering process maturity model consists of three levels. The first two levels are comparable to the first two level in the CMM. The last level, Level 3 - Defined, is comparable to the other three remaining levels (from Level 3 to Level 5) in the CMM. With respect to the CMM, the requirements engineering maturity model further defines the maturity of requirements engineering practice. Although the CMM involves several requirements engineering activities, it provides limited support to define requirements engineering maturity based on industry standards and practice [Linscomb, 2003].

Unfortunately, any software process tends to concentrate effort on requirements at the start of any project. This limits any understanding of requirements evolution and makes it hard to recover good quality data on requirements evolution.

2.1.2 Empirical Requirements Evolution

Empirical analyses of requirements evolution (generally speaking, of requirements engineering methodologies and practices) are still too patchy. Even the empirical validation of requirements engineering methodologies is still a fertile and challenging research as well as practice field. The comparison between related requirements engineering methodologies lacks of any prominent analysis. On one hand it is difficult to collect and analyse long-term data about requirements evolution. The life cycles of many software systems span several years or even decades. On the other hand requirements engineering methodologies currently provide limited support in order to analyse requirements evolution [Jarke and Pohl, 1994, van Lamsweerde, 2000]. Practitioners perceive that analysis and collection of long-term data just overload their requirements practices. In spite of this set of circumstances, recent empirical analyses of requirements evolution provided encouraging results.

Measuring requirements (quality) [Hooks and Farry, 2001] represents the basis for improvement. Most of the time managers believe that measuring requirements is too expensive (in terms of resources) and time consuming, despite requirements engineering being the systems development activity with the highest return on investment payoff [Van Buren and Cook, 1998]. Others think that it is possible to measure requirements only after project completions. The main motivation of measuring requirements is to identify opportunities for improvement. It is possible to measure the quality of requirements on data drawn from Change Requests (CRs) and Discrepancy Reports (DRs) [Hooks and Farry, 2001]. These are mainly different names for stimulus for changes into requirements. Simple measures of requirements (e.g., statistics derived from the analysis of requirements counting, trends, percentages, as well as classifications) can be very effective. These metrics represent a basis for *Changes Analysis*. Moreover, they represent a means for comparing projects. A systematic account of requirements measures provides input for improving and changing organisation requirements processes. Many issues in changing requirements processes are due to the lack in requirements culture within organisations, lack of empirical evidences, as well as difficulties in identifying stable (or volatile) requirements. Engineering tools and analysis models have to support any shift in requirements processes.

New requirements can arise from operational anomalies [Lutz and Mikulski, 2003]. Although the resulting requirements may be incomplete, requirements evolution can capture rare environmental events. On one hand very rare events may raise anomalies. On the other hand requirements evolution can be a strategy for dealing with unforeseen critical events. New requirements can enhance system features (e.g., fault tolerance) in order to tackle corelated failures (e.g., hardware failures) [Lutz and Mikulski, 2003]. For instance, software requirements may evolve in order to compensate hardware degradation [Lutz and Mikulski, 2003]. Requirements changes may therefore propagate through holistic (or systemic) dependencies. Monitoring these dependencies may provide further information about requirements evolution. Requirements evolution may therefore provide diverse strategies dealing with environmental changes (e.g., frequent policy changes, rare events, hardware degradations, etc.). Release management and cost assessment may further benefit from analyses of requirements evolution [Stark et al., 1998]. These empirical results point out that current requirements engineering or maintenance analyses provide limited support to understand requirements evolution. This is because maintenance methodologies tend to focus on classifying and managing requirements changes, rather that on analysing or anticipating the changes [Lutz and Mikulski, 2003]. Note that evolution is different than reuse. Requirements reuse should take into account a trade off between process and product viewpoints

[Lam, 1997, Lam et al., 1997]. Reusing requirements has to be carefully adapted to specific software contexts [Lam, 1997, Lam et al., 1997].

Other studies [Hammer et al., 1998] identify poor requirements baselines as major causes for requirements evolution. Although poor requirements baselines trigger only specific types of requirements changes, it justifies the up front effort in writing quality requirements specifications. Guidelines [Sommerville and Sawyer, 1997a] and quality requirements specifications [Alexander and Stevens, 2002, Lauesen, 2002] may reduce changes that increase rework on requirements [Hammer et al., 1998].

Despite the complex nature of requirements evolution, the reported results have effectively used very simple empirical analyses. One common practice across the different case studies is the classification of requirements changes. On one hand classifications represent the basis of software measurement [Fenton and Pfleeger, 1996]. They are very simple tools that support analyses throughout the software life cycle. On the other hand classifications evolve themselves due to work practice. Moreover it is difficult to compare classifications across different and diverse contexts. Thus, it is easy to construct classifications within specific domains. Although classifications are strongly related to their origins [Bowker and Star, 1999].

It is possible to analyse requirements evolution by looking at simple classifications of requirements and requirements changes. The PROTEUS project [Harker et al., 1993, PROTEUS, 1996] classifies requirements as *stable* or *changing*. Table 2.1 shows the PROTEUS classification that consists of six types of requirements, i.e., STABLE, CHANGING: Mutable, Emergent, Consequential, Adaptive and Migration. This simple classification stresses two important aspects of requirements evolution. The first is that stable requirements do exist. The second is that each type of requirement originates from different sources within software environments (e.g., business core, environmental turbulence, etc.). The PROTEUS project moreover identifies other dimensions of change[2]. Each dimension describes a different aspect relevant to requirements evolution. The PROTEUS project further elaborates the analysis of requirements evolution by identifying the major problem areas associated with changes. The PROTEUS

[2]Source of change; chronology and rate of change; impact of change; risk of change; implementation of change: representation of change, communication regarding change, recording change, validation of change, methodology of change, cost of change; speed of implementing change; organisational culture.

Table 2.1: The PROTEUS classification of types and origins of changing requirements.

Type of requirement	**Origins**
STABLE	Technical core of the business
CHANGING	
Mutable	Environmental turbulence
Emergent	Stakeholder engagement in requirements elicitation
Consequential	System use and user development
Adaptive	Situated action and task variation
Migration	Constraints of planned organisational development

project identifies ten problem areas (i.e., requirements engineering process, project management, contractual boundaries, change process, standards, assessment of impact, communication, levels of specification, traceability and documentation) associated with changing requirements. The analysis of these problems points out two things. The first is that changes occur for diverse reasons, which are unrelated to “errors” in the specification process. Thus, any sound specification process may experience requirements changes. The second is that changes may themselves be unproblematic, but the consequences of changes may cause problems.

Table 2.2 shows another classification (very similar to the PROTEUS classification) of volatile requirements [Sommerville and Sawyer, 1997a]. The requirements classification provides a strategy in order to simplify requirements change management. Moreover, system design should take into account information about volatile requirements. For instance, loosely coupled modules may easily accommodate requirements changes. Requirements classifications help to identify volatile requirements and to plan for likely changes [Sommerville and Sawyer, 1997a]. Knowing the likelihood of requirements changes supports change management. General system requirements are very expensive to change (e.g., architecture changes) and may involve different stakeholders. Change management policies and strategies may help to isolate volatile requirements from stable ones . The definition of change management policies is important in order to have a formal mechanism to deal with changes in requirements. Empirical analyses have to support change management policies. A systematic

Table 2.2: Types of volatile requirements.

Type of requirement	**Description**
Mutable requirements	These requirements change because of changes to the environment in which the system is operating.
Emergent requirements	These requirements cannot be completely defined when the system is specified but which emerge as the system is designed and implemented.
Consequential requirements	These requirements are based on assumptions about how the system will be used. When the system is put to use, some of these assumptions will be wrong. Users will adapt the system and find new ways of use its functionalities. This will result in demands from users for system changes and modifications.
Compatibility requirements	These requirements depend on other equipment or processes. As these change, these requirements also evolve.

methodology to analyse requirements evolution helps to trace requirements changes and to assess the impact of change. Unfortunately, the volume of information related to requirements evolution represent a major issue in any change management policy. Simple instruments like maintaining the history of change may become impractical [Lauesen, 2002, Robertson and Robertson, 1999]. After a while, the list of changes is so huge that it is difficult to realise what is going on without a systematic method to shape angles for analysing evolutionary information. For instance, The classification of requirements changes or the allocation of requirements to system functions effectively support the analysis of evolutionary data. It is therefore possible to narrow the focus of the analysis in order to identify evolutionary properties.

The analysis of the origins of changes may produce other classifications relevant to requirements evolution. Table 2.3 shows a classification of change factors for requirements [Kotonya and Sommerville, 1996]. It is important to note that all these classifications capture diverse socio-technical relationships. That is, socio-technical interactions drive system evolution, hence, requirements evolution.

Table 2.3: Factors leading to requirements changes.

Change factor	Description
Requirements errors, conflicts and inconsistencies	As requirements are analysed and implemented, errors and inconsistencies emerge and must be corrected. These problems may be discovered during requirements analysis and validation or later in the development process.
Evolving customer/end-user knowledge of the system	As requirements are developed, customers and end-users develop a better understanding of what they really require from a system.
Technical, schedule or cost problems	Problems may be encountered in implementing a requirement. It may be too expensive or take too long to implement certain requirements.
Changing customer priorities	Customer priorities change during system development as a result of a changing business environment, the emergence of new competitors, staff changes, etc.
Environmental changes	The environment in which the system is to be installed may change so that the system requirements have to change to maintain compatibility.
Organisation changes	The organisation which intends to use the system may change its structure and processes resulting in new system requirements.

2.1.3 Modelling in Requirements Engineering

Modelling has attracted a substantial effort from research and practice in requirements engineering. In spite of quality and effective development processes, many faults in software systems are traced back to high level requirements. This has motivated the increasing use of modelling in requirements engineering. The aim of requirements modelling is twofold. On one hand modelling contributes towards *correctness* and *completeness* of requirements. On the other hand modelling supports *validation* and *verification* of requirements. The overall goal of modelling is mainly to reduce the gap between system requirements and design. Modelling tackles two main requirements issues. The first is that translations from requirements to design are error-prone. The second is that stakeholders (e.g., system users, system engineers, etc.) have often contradicting understandings about requirements. These problems have motivated the blossom of many modelling methodologies and languages (e.g., UML [Rumbaugh et al., 1999]) used in practice. The requirements-design gap has been believed to be the major source of requirements changes. Although this gap is one of the sources of requirements changes, research on requirements evolution (e.g., [PROTEUS, 1996]) clearly points out other origins of changes.

Modelling[3] incorporates design concepts and formalities into requirements specifications. This enhances our ability to assess requirements correctness and completeness. For instance, *Software Cost Reduction*[4] (SCR) consists of a set of techniques for designing software systems [Heitmeyer, 2002]. The SCR techniques support the construction and evaluation of requirements [Heitmeyer et al., 1998]. The SCR techniques use formal design techniques, like tabular notation and information hiding, in order to specify and verify requirements. According to information hiding principles, separate system modules have to implement those system features that are likely to change. Although module decomposition reduces the cost of software development and maintenance, it provides limited support for requirements evolution. SCR therefore provides limited mechanisms to deal with requests of requirements changes [Wiels and Easterbrook, 1999], hence requirements evolution.

Intent Specifications [Leveson, 2000] further support the analysis and design of evolving systems. The Intent Specifications consist of five different hierarchical levels[5]. Each level provides rationale (i.e., the intent or "why") about the level below. Each level has mappings that relate the appropriate parts to the levels above and below it. These mappings provide traceability of high-level system requirements and con-

[3] Recent research is pursuing further results in model-driven software development. Model-driven software development supports the automation of development activities (e.g., verification and validation) by techniques like model-checking or theorem proving. Among relevant research subjects are, for instance, model-driven development (e.g., [Weiss et al., 2003]), model-driven testing (e.g., [Gargantini and Heitmeyer, 1999]) and human factors modelling (e.g., [Rushby, 2002]).

[4] David L. Parnas developed some of the SCR underlying techniques. A collection book contains the most relevant work by Parnas [Hoffman and Weiss, 2001].

[5] Level 1, system purpose; Level 2, system principles; Level 3, blackbox behavior; Level 4, design representation; Level 5, physical representation or code. Note that a recent version of Intent Specifications [Weiss et al., 2003] introduces two additional levels: Level 0 and Level 6. Level 0, the management level, provides a bridge from the contractual obligations and the management planning needs to the high-level engineering design plans. Level 6, the system operations level, includes information produced during the actual operation of the system.

straints down to physical representation level (or code) and vice versa. In general, the mappings between Intent levels are many-to-many relationships. In accordance with the notion of *semantic coupling*, Intent Specifications support strategies[6] to reduce the cascade effect of changes [Weiss et al., 2003]. Although these strategies support the analysis and design of evolving systems, they provide limited support to understand the evolution of high-level system requirements[7]. The better our understanding of requirements evolution, the more effective design strategies. That is, understanding requirements evolution enhances our ability to inform and drive design strategies. Hence, evolution-informed strategies enhance our ability to design evolving systems.

Modelling methodologies and languages advocate different design strategies. Although these strategies support different aspects of software development, they originate in a common *Systems Approach*[8] to solving complex problems and managing complex systems. In spite of common grounds, modelling methodologies and languages usually differ in the way they interpret the relationships among heterogeneous system parts (e.g., hardware components, software components, organisational components, etc.). For instance, the SCR requirements method [Heitmeyer et al., 1998] relies on the *Four Variable Model* [Parnas and Madey, 1995]. The Four Variable Model describes the system requirements in terms of mathematical relations over environmental quantities. Environmental quantities can be either *monitored* or *controlled*. The system measure the monitored quantities, whereas it control the controlled ones. Sets of

[6]Possible strategies are: (1) eliminating tightly coupled (i.e., many-to-many) mappings (2) minimising loosely coupled (i.e., one-to-many) mappings.

[7]Leveson reports the problem caused by "Reversals" in TCAS (Traffic Alert and Collision Avoidance System) [Leveson, 2000]: "About four years later the original TCAS specification was written, experts discovered that it did not adequately cover requirements involving the case where the pilot of an intruder aircraft does not follow his or her TCAS advisory and thus TCAS must change the advisory to its own pilot. This change in basic requirements caused extensive changes in the TCAS design, some of which introduced additional subtle problems and errors that took years to discover and rectify."

[8]The essays collected in [Hughes and Hughes, 2000] give an historical account of the Systems Approach: "Practitioners and proponents embrace a holistic vision. They focus on the interconnections among subsystems and components, taking special note of the interfaces among various parts. What is significant is that system builders include heterogeneous components, such as mechanical, electrical, and organizational parts, in a single system. Organizational parts might be managerial structures, such as a military command, or political entities, such as a government bureau. Organizational components not only interact with technical ones but often reflect their characteristics. For instance, a management organization for presiding over the development of an intercontinental missile system might be divided into divisions that mirror the parts of the missile being designed.", INTRODUCTION, p. 3.

measured and controlled quantities define the basic relations: NAT, REQ, IN, OUT, and SOF. The relation NAT defines environmental constraints on the system behaviour. The relation REQ defines system constraints on the environmental quantities. Whereas, the relations IN and OUT define the mappings between the monitored variables and the input devices, and between the output devices and the controlled variables respectively. Finally, the relation SOF defines the behaviour of the software system. These relations describe the software system behaviour as a whole, hence the software requirements specification (SOFTREQ) [Heitmeyer et al., 1998].

Similarly, a reference model for requirements and specifications defines system and environment in terms of basic artifacts [Gunter et al., 2000]. The reference model, called the WRSPM model, consists of five artifacts: *World* (W), *Requirements* (R), *Specification* (S), *Program* (P) and *Machine* (M). The World artifact identifies the domain knowledge that provides presumed environment facts. The Requirements artifact indicates what the customer needs from the system, described in terms of its effect on the environment. The Specification artifact provides enough information for a programmer to build a system that satisfies the given requirements. The Program implements the specification using the specific programming platform in the Machine artifact. The Machine artifact provides the basis for programming a system that satisfies the requirements and specification. The Word, Requirements and Specification artifacts pertain mostly to the environment. Whereas, The Specification, Program and Machine artifacts pertain mostly to the system. In the WRSPM model, the Specification artifact has a central position between the environment and the system. Whereas, the Requirements artifact stands between the World and the Specification artifacts.

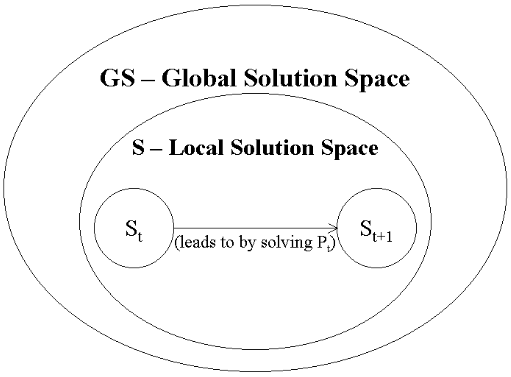
The Intent Specifications extend over three dimensions [Leveson, 2000]. The vertical dimension consists of five (or seven [Weiss et al., 2003]) hierarchical levels that represent the intent. Along the horizontal dimension, the Intent Specifications decompose the whole system in heterogeneous parts: Environment, Operator, System and Components. The third dimension, *Refinement*, further breaks down both the Intent and Decomposition dimensions into details.

This type of model (e.g., Four Variable Model, WRSPM model and Intent Specifications) represents software systems in terms of basic parts (e.g., environmental quantities, artifacts, intents, etc.). Although these models give different representations, the Systems Approach represents a common origin for all of them. A common aspect is that models identify the relations between the different system parts. On one hand these relations constrain the system behaviour (e.g., by defining environmental dependencies). On the other hand they are very important for system management and design. Among the different relations over heterogeneous system parts and hierarchical levels is *Traceability*. Requirements traceability consists of emergent heterogeneous system information providing a basis for management throughout software development. Traceability identifies a multidimensional [Jarke, 1998] structure that enhances requirements management. Among the dimensions of traceability are *pre-traceability* and *post-traceability* [Gotel and Finkelstein, 1994]. Pre-traceability points to requirements sources, whereas post-traceability points to those artifacts (e.g., other deliverables) that are related to requirements [Gotel and Finkelstein, 1994]. Although traceability supports management, traceability often faces many issues in practice [Gotel and Finkelstein, 1994, Ramesh, 1998]. In particular, traceability faces evolution [De Michelis et al., 1998]. With respect to traceability, requirements evolution therefore identifies another system dimension.

Looking at requirements from a heterogeneous engineering [Bijker et al., 1989] perspective further explains the complex interaction between system (specification) and environment. The most common understanding in requirements engineering considers requirements as goals to be discovered and (design) solutions as separate technical elements. Hence requirements engineering is reduced to be an activity where technical solutions are documented for given goals or problems. Differently according to heterogeneous engineering, requirements specify mappings between problem and solution spaces [Bergman et al., 2002a]. Both spaces are socially constructed and negotiated through sequences of mappings between solution spaces and problem spaces. Figure 2.6 shows a representation of the *Functional Ecology* model, which implies that requirements emerge as a set of consecutive solution spaces justified by a problem space of concerns to stakeholders [Bergman et al., 2002b]. This view defines evolutionary cycles of iterations in the form:

$$solution \rightarrow problem \rightarrow solution\ .$$

This implies that requirements engineering processes consist of solutions searching for problems, rather than the other way around (that is, problems searching for solutions).



Space	Description
S - Local Solution Space	A Local Solution Space is the current solution space, and all locally accessible solution spaces that can be reached from the current solution space using available skills and resources offered by the principals.
GS - Global Solution Space	A Global Solution Space is the space of all feasible solution spaces, including those not currently accessible from the local space, that require mobilisation of all principals and technologies to be accomplished. All local solution spaces exists within relevant global solution space. That is, S is a subspace of GS.
S_t - Current Solution Space	The Current Solution Space embodies the history of solved social, technical, economical and procedural problems that constitute the legacy of previously solved problems at current time t. The current solution space exists within a local solution space. That is, S_t is a subspace of S.
P_t - Proposed System Problem Space	A Proposed System Problem Space is the space that contains all the recognised problem chosen by the principals at time t that justify the proposed system.
S_{t+1} - Proposed Solution Space	A Proposed Solution Space, or simply a Proposed Solution, is a subspace of the future solution space S_{t+1} that includes the reconciliation of S_t to P_t, i.e., $S_t \rightarrow P_t \rightarrow S_{t+1}$.

Figure 2.6: Functional ecology solution space transformation.

The main difference between the solution space transformation and other requirements engineering models is its emphasis on stakeholder interactions. Requirements, as mappings between solutions and problems, represent an historical account of solved socio-technical problems within industrial settings. Requirements therefore capture the social shaping [MacKenzie and Wajcman, 1999] of systems. This heterogeneous account of requirements is convenient to capture requirements evolution.

2.1.4 Modelling Requirements Evolution

Despite the existence of many modelling methodologies and languages, very few address requirements evolution. The PROTEUS Project [PROTEUS, 1996] proposes a formal framework for representing and reasoning about requirements changes. The formal representation consists of a *goal-structures framework*. Table 2.4 describes the fundamental components of the goal-structures framework: *goals*, *effects*, *facts* and *conditions*.

Table 2.4: The basic components of the PROTEUS goal-structures framework.

Component	**Description**
Goal	A goal is a statement, or assertion, identifying a property that the system has to comply with. Each goal has an associated *strategy* that describes how that goal may be achieved.
Effect	An effect is similar to a goal that may need further decomposition or explanation. Effects may be goals, which may be unachieved (or unnecessary).
Fact	A fact is a true statement (or assumption about property of the system).
Condition	A condition, like facts, are basic undecomposable statements. Differently than facts, conditions may be false. Consistent sets of conditions represent *scenarios* to analyse the behaviour of the system (under different sets of conditions).

The facts in some scenario should support or satisfy the decomposition of goals. In this way, the set of facts represents a model of the system being developed, a scenario represents the input to that model, and the goals represent requirements on the output, or behaviour, of that system. The specification of strategy for a goal, along with other goals and facts expressed in the model, means that the goal would be satisfied. This is a *local support* for the goal, because there is an assumption that any other goal referred to can be achieved through further decomposition. The decomposition of goals continue until all goals are reduced to facts (or conditions). The model alone will provide a *global support* for every goal in the structure. In practice a strategy will need to provide a solution to multiple goals.

The PROTEUS goal-structures framework represents requirements and their interactions with respect to requirements changes. Most importantly, the framework captures the interactions between system and the environment in which it operates. These interactions (i.e., between requirements, and between system and environment) form a basis for sensitivity and impact analyses. In order to be effective, these analyses have to take into account information about requirements volatility and rationale for design decisions. The formal representation furthermore supports reasoning on requirements changes. It supports *sensitivity analysis* and *impact analysis* of requirements changes. In order to assess the risk associated with requirements changes it is necessary to assess the likelihood of requirements changes together with the impact of changes. The combination of sensitivity and impact then provides a measure of risk [PROTEUS, 1996]. Table 2.5 shows the information framework required to perform sensitivity and impact analyses [PROTEUS, 1996]. The information framework identifies the input to the analyses, how to acquire evolutionary information and the output of the analyses.

Table 2.5: Information framework for sensitivity and impact analyses.

Sensitivity Analysis		
Input	**Information Gathering**	**Output**
Knowledge of external factors that could cause requirements to changes. Past history of changes. Experience of change. Requirements classifications. Experience of past history of design sensitivity to changing requirements.	From: documentation, worst case scenarios, brainstorming, reviews and checklists.	List of requirements most likely to change, with likelihood in what way, by how much and when in life cycle. List of design areas most susceptible to changes.

Impact Analysis		
Input	**Information Gathering**	**Output**
In what way and by how much the requirements changes. Where the change impacts. Design rationale based on requirements. What other requirements are affected. When change is likely to occur.	From sensitivity analysis. Traceability between requirements and design area. Recording and traceability of design rationale. Traceability between requirements.	Estimation of impact on design. Indication of conflict and ripple effects. Estimation of impact on cost and other objectives.

Finally, the framework represents a means of communication in the development environment. The *formal* and *informal* representations of requirements allow easy communication and translation of requirements as well as requirements changes. Although the goal-structures framework provides an alternative representation of requirements, it has many similarities to other requirements models (e.g., the Four Variable Model).

Modelling requirements evolution can unfold according to two general strategies. The first is to model requirements in such a way that they can easily evolve. The second is to model evolution itself into requirements models. The latter is the underlying idea of modelling the evolution of artifacts [Rolland, 1994]. The *Evolutionary Object Model* [Rolland, 1994] captures design history. Thus, each evolutionary object is a representation of its own historical evolution[9].

Another strategy is directly to model requirements evolution. One way of modelling requirements evolution is by ordered sequences of requirements releases. Requirements therefore evolve by changes from one release to the successive one. This is the intuitive explanation of the idea behind the logical framework [Zowghi et al., 1996, Zowghi and Offen, 1997] for modelling and reasoning about requirements evolution. The logical framework relies on two basic operations: a *nonmonotonic inference*, $|\!\sim$, and a *belief revision*, $*$. Figure 2.7 shows how these operations combined capture requirements evolution. The nonmonotonic inference, $|\!\sim$, completes the requirements model by assuming defaults within the specific domain model. The initial incomplete set of requirements is completed by applying relevant defaults or tentative assumptions about the problem domain. There could be possibly multiple, mutually contradictory ways of completing the requirements model[10]. Stakeholders (e.g., system users, requirements engineers, etc.) will choose how to complete the requirements model. Whereas, the belief revision, $*$, maps (nonmonotonic) requirements models in order to obtain a revised nonmonotonic theory representing the new requirements model. The underlying theory of belief change provides a semantic basis for rational belief revision[11]. Thus, requirements evolution consists of completing requirements

[9]The classification of object evolution consists of three categories: *transformation*, *expansion* and *mutation*. A transformation affects the object definitions (e.g., renaming the object). An expansion changes the object spatial environment that results from the relationships with other objects (e.g., introducing new relationships with other objects). A mutation introduces links between different types of objects.

[10]Sets of nonmonotonic consequences or extensions complete the nonmonotonic requirements model.

[11]The underlying theory formalise the process of revising beliefs. The *principle of minimal change* (or *criterion of information economy*) would retain as much as possible old beliefs in future time (or state). The belief framework provides three operations: *Expansion*, *Contraction* and *Revision*.

(by nonmonotonic inferences) and refining them (by belief revisions).

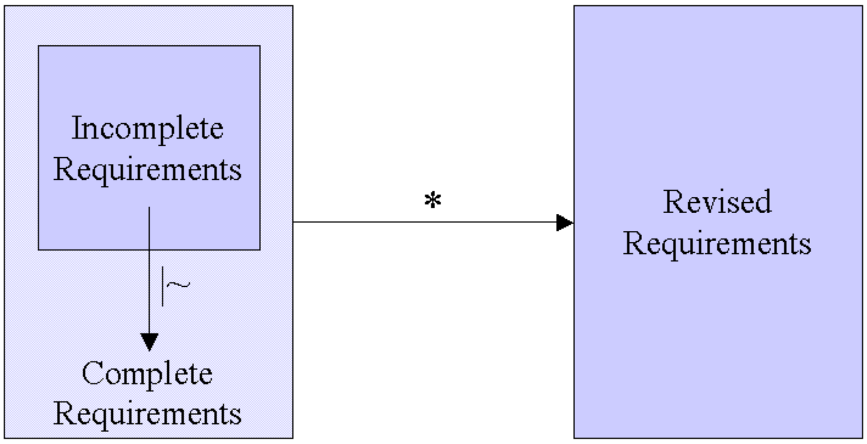


Figure 2.7: Modelling requirements evolution.

There is little coherence among the different models that address requirements evolution. On one hand this is due to the complexity of requirements evolution. On the other hand requirements evolution has received little attention. Although the evolutionary models capture diverse aspects, it is possible to identify stakeholder interactions as an important driver of evolution.

2.2 Towards Requirements Evolution Engineering

Requirements engineering research and practice highlight evolution as an important aspect of software production. Although research results and experience are still patchy, requirements evolution emerges as a comprehensive viewpoint that allows us to further understand the mechanisms of socio-technical system evolution. On the other hand requirements evolution provides new insights in research and practice in software production.

Iterative development processes emphasise evolutionary aspects of software production. Although management and development processes provide overall organisation in terms of development activities and phases, they require to be tailored for specific software systems and design contexts. Empirical analyses allow us to understand evolutionary aspects of software production. Requirements evolution provides new grounds for understanding the mechanisms of socio-technical system evolution. Modelling requirements (evolution) captures the understanding of software system evolution. Although requirements evolution emerges as an important aspect of software production, requirements engineering tools[12] provide limited support for the analysis of requirements evolution. Requirements management tools mainly support requirements (change) management based on traceability. For instance, requirements management tools (e.g., Telelogic DOORS® and IBM Rational® RequisitePro®) rely on traceability in order to assess the impact of requirements changes. Most requirements tools fall dawn into two categories: requirements and changes management tools. Other tools support specific requirement analyses (e.g., The NASA Automated Requirements Measurement, ARM, Tool provides measures that can be used to assess the quality of requirements specification documents).

By contrast, this thesis takes requirements evolution as inherent in software production. It investigates the current understanding of requirements evolution and explores new directions in requirements evolution research. The empirical analysis of industrial case studies highlights software requirements evolution as an important issue. Unfortunately, traditional requirements engineering methodologies provide limited support to capture requirements evolution. This thesis highlights a set of methodologies towards *Requirements Evolution Engineering* (REE). This thesis addresses the problem of empirically understanding and modelling requirements evolution.

[12] The INCOSE (International Council on Systems Engineering) Tools Database Working Group conducted a survey on Requirements Management Tools. The questionnaire used for the survey takes into account different aspects (e.g., capturing system element structure, traceability analysis, etc.) of requirements tools. The Atlantic Systems Guild Inc. provides another review of Requirements Tools.

Chapter 3

An Avionics Case Study

This chapter describes the investigation of an avionics safety-critical industrial case study with respect to requirements evolution. The investigation of an industrial case study allows us to acquire input from practice in requirements engineering. This is to take realistic account of requirements evolution. Moreover, a case study provides domain knowledge that characterises the specific industrial context. Case studies drawn from industry stress the crucial aspect of domain knowledge. Most of requirements engineering methodologies have serious practical limitations, because they lack of domain knowledge.

3.1 Description of the Case Study

The case study consists of software that satisfies the most stringent level of the standard DO178B [RTCA, 1992]. Two main business stakeholders cooperate in the definition of the system requirements. The *customer*, who provides the general system requirements, and the *supplier*, who produces the software. The remainder of this section describes features of the case study that are relevant to requirements evolution. The description is twofold: the system requirements and the development process.

3.1.1 System Requirements

The investigation of the case study takes into account the system software requirements. They have been drawn from the general system requirements provided by the customer. The definition of the software requirements involves both the customer and the supplier in elicitation, specification and negotiation. The supplier is responsible for producing the system software and its compliance with the specified software requirements. The system software requirements fall into two main categories: Safety Requirements; Functional and Operational Requirements.

Safety Requirements. A system safety assessment analyses the system architecture to identify and classify the failure conditions. Safety related requirements are then identified and allocated to the software or hardware requirements. The allocation of requirements to software or hardware can be unclear at some stage of the development life cycle. Thus there are cases in which the allocation is delayed until further information is available correctly to allocate requirements.

Functional and Operational Requirements. The customer provides the system requirements that contain information needed to describe the functional and operational software requirements. These include timing and memory constraints and accuracy requirements where applicable. Requirements also contain details of inputs and outputs the software has to handle with special emphasis to those using non-standard data formats. As for the safety requirements if the allocation of requirements to software or hardware is unclear any decision is delayed until further notice.

3.1.2 Development Process

The software development process of the case study is an instance of the V model[1], which emphasises relations between the design and coding phases with that of (inte-

[1]The V model is one of the many models (e.g., Waterfall, V model, Spiral, etc.) that have been proposed in software engineering. The most common software development processes are introduced in any general software engineering textbook, e.g., [Pfleeger, 1998, Sommerville, 2001]. Most of them agree that software development is iterative. Hence software as well as software requirements can only be specified in subsequent releases, which emphasise the evolutionary nature of software as well as software requirements.

gration as well as system) testing, verification and validation. The V model implies that if problems arise during verification and validation, they are reported and eventually fixed by refinements and corrections in the requirements and design as well as in the implementation. Then the verification and validation phases are repeated again to reassess the software system. The development process points out the iterative nature of designing and implementing software systems. The remainder of this section describes the software development process and its iterative aspects.

Software Development Process. Figure 3.1 shows the development process of the case study. 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.

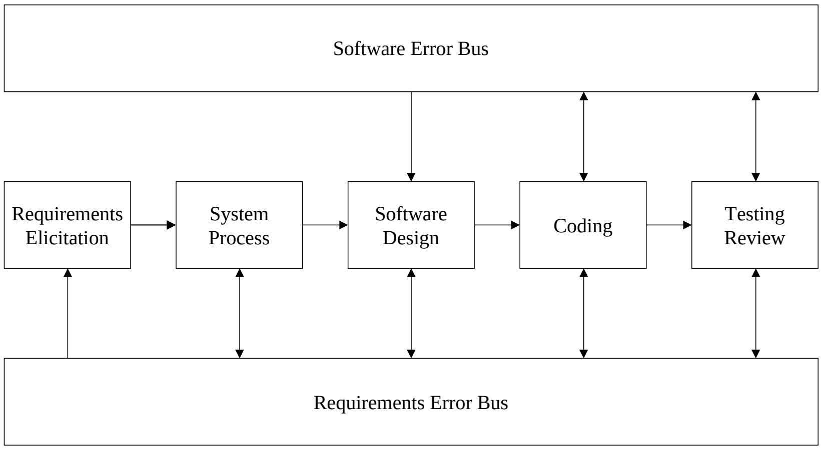


Figure 3.1: The safety-critical software development process of the case study.

The subsequent development phases, from Requirement Elicitation to Testing and Review, point out issues and changes, which give rise to feedback loops in the development process. Reported issues fall into two main categories, depending on whether or not they affect the requirements or just the design and implementation of the system. Figure 3.1 shows them as two different error busses: *Software Error Bus* and

Requirements Error Bus. Notice that as the software development progresses from requirements to design and code, problems require to modify deliverables of the previous phases (in other words, changes move backwards in the development process). The supplier has defined only general guidelines dealing with changes, because the size and scope of these changes differ for each modification and for each project. 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 to modify the original design. Moreover, issues, hardware changes or even stakeholder changes may give rise to requirements changes at any stage of the software project.

As design progresses the partition and refinement of software requirements cause an expansion of information, which extend through subsequent activities (e.g., testing) and across deliverables (e.g., code). Activities as well as deliverables reflect the resulting fragmentation. For example, the coding phase consists of a collection of interrelated implementing phases. The boundaries between these phases may overlap due to unclear process definitions or implementation's dependencies. Thus the integration of some code items may begin before the implementation of the remainder is completed. This is often the case, even in small simple projects. It is therefore very difficult to identify and define the limits of any phase and the transitions between phases. Thus elements within each phase may be at different progress stages. A strict configuration management policy requires to maintain traceability in order to ensure that the final code is verified and it is complete and consistent with its high level requirements.

Software verification mainly consists of testing and review analysis. Specific documents, produced at the beginning of the project, identify to which extent software verification is required at each phase of the development process. Changes affecting certified software require to repeat the entire software verification. Whereas requirements changes require to iterate the entire development process. The extent to which all phases comply with the required changes depend on the time elapsed since certification. If changes occur during the period between flight trials and certification, project documents are required to be consistent with the implemented software. Hence, reviews need to comply with occurring changes depending on the size of the modifications.

Requirements Documentation. 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* consist of the requirements identified for the whole systems without distinguishing between software and hardware requirements. The *System Process* translates the System Requirements in the *Software Functional Requirements*. The Software Functional Requirements consist of a set of documents that specify the requirements for each software function identified for the system. These software functions will then be designed, implemented and integrated by subsequent activities in the development process.

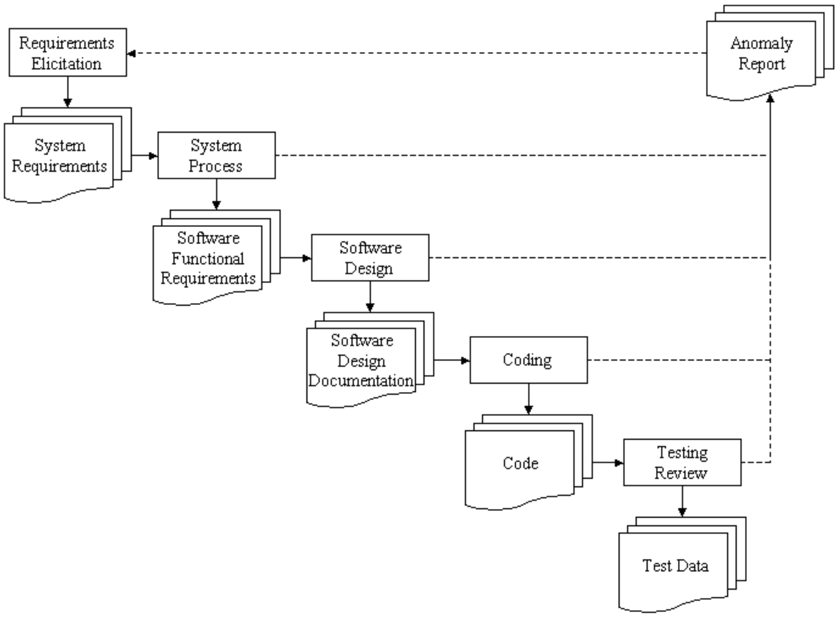


Figure 3.2: Development activities and deliverables.

Anomalies (e.g., faults, failures, misbehaviours, etc.) encountered during development are reported by *Anomaly Reports*. Anomaly Reports contain all information that may be useful to the development team in order to assess possible faults. For each recognised fault, needed changes are allocated to specific software releases in order to fix it. The scope of these changes ranges from requirements to software code. Thus, Anomaly Reports continuously provide feedback. Notice that certification requires to trace all changes.

Product Line Aspects and Standards. Hardware dependent software requirements arise not directly from the system requirements, but from the implementation of those system requirements. This is normally due to the way hardware has been designed to meet its requirements as well as an indirect result of safety related requirements. The hardware dependent software requirements characterise the specific product-line in terms of hardware constraints and safety requirements. Certification authorities assess mainly by a certification plan whether the proposed software development process is commensurate with the proposed software level. The plan of the case study complies with the specific guidelines in the standard DO-178B [RTCA, 1992].

3.2 Empirical Investigation

The case study consists of software for an avionics system. The supplier maintained a data repository in order to trace requirements changes. Requirements changes occurred due to the Anomaly Reports during the projected. This section describes the empirical investigation of the case study. The investigation consists of diverse empirical analyses of the data repository. The empirical analyses aim to identify requirements properties that may enhance our understanding of requirements evolution [Anderson and Felici, 2000a, Anderson and Felici, 2000b]. The investigation moves from a general viewpoint towards a product-oriented viewpoint. The lack of any well established methodology to analyse requirements evolution denies us any preferred analysis to commence our investigation. Hence we adopt an incremental strategy in order to effectively conduct our investigation. The incremental strategy furthermore allows us to refine our investigation into subsequent empirical analyses. The remainder of this section describes all the empirical analyses conducted on the case study.

3.2.1 Requirements Evolution

The initial empirical analysis of the avionics case study looks at simple requirements trends, which are extrapolated from the data repository of requirements changes. There are 22 successive releases of the software requirements specification, each one corresponding to a software release[2]. Figure 3.3 shows the total number of requirements changes, i.e., *Added, Deleted and Modified Requirements*, over the 22 software releases. We have been able to draw Figure 3.3, because each requirements change was uniquely identified in the data repository. Furthermore, the configuration management policy within the project required to uniquely identify the releases of the software requirements specification as well as the software code. The data repository consists of the allocation of specific requirements changes to specific software requirements specification releases.

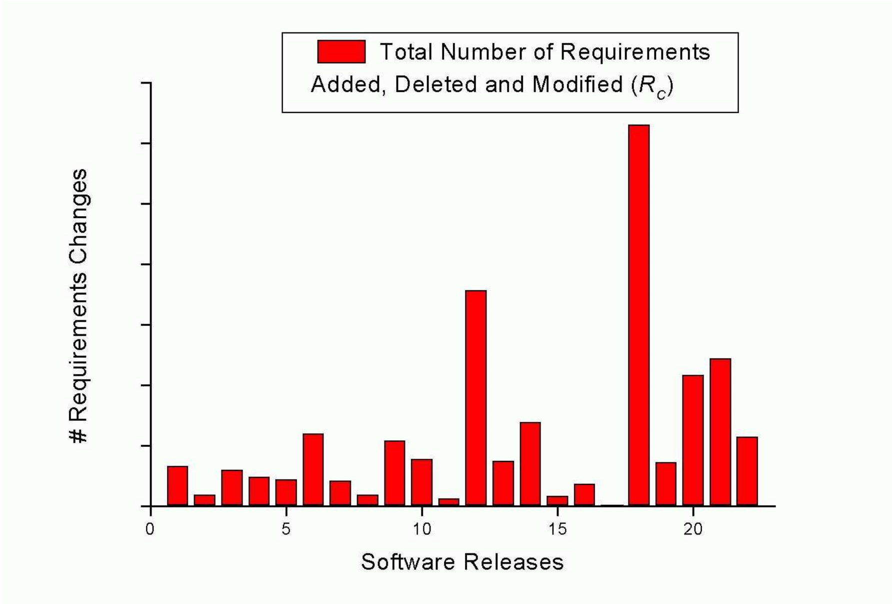


Figure 3.3: Number of requirements changes per software release.

The trend of requirements changes provides little information about evolutionary features of requirements, but it emphasises requirements evolution. Figure 3.4 shows that the size (in terms of number of requirements) of the requirements specification

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

rises over the software releases. The increasing trend points out that there is a predominance of new added requirements into requirements changes. This is also confirmed by looking at the requirements changes repository.

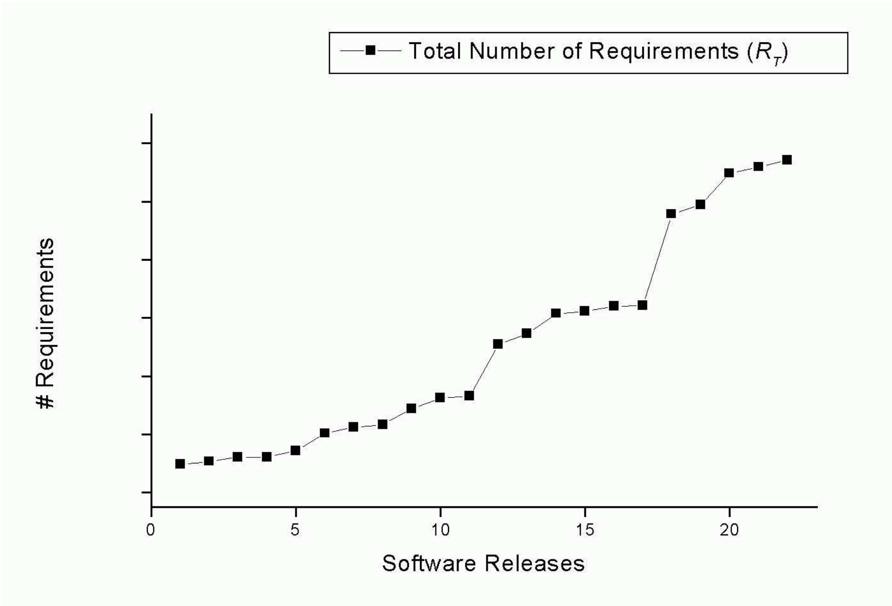


Figure 3.4: Total number of requirements per software release.

The increasing number of requirements is due to stakeholders who better understand system requirements and refine them into smaller and more precisely stated requirements. Another reason is that stakeholders discover new requirements that were missing at the beginning due to lack of information and requirements dependencies. Finally, design, implementation and testing activities provide further feedback into the requirements.

3.2.2 A Taxonomy of Requirements Changes

Changes affect requirements attributes like variables, functionality, explanation, traceability and dependency. These attributes are usually embedded within paragraphs specifying requirements. The use of requirements templates can effectively be a way to collect and represent requirements (e.g., [Robertson and Robertson, 1999]). A structured way of representing, collecting and organising requirements attributes may be useful to identify information for controlling and monitoring requirements evolution.

The inspection of the history of changes points out the types of changes that have occurred in the requirements. Table 3.1 shows a taxonomy of requirements changes in the case study.

Table 3.1: Types of changes identified by inspection of the history of changes.

Type of Change	**Description**
Add, Delete and Modify requirements	Each requirements change can be traced to the basic activities, i.e., adding, deleting and modifying requirements, of changing the requirements documents.
Explanation	Some requirements may need further explanation in order to minimise misunderstandings. The explanation paragraphs of requirements are changed for clarity.
Rewording	Requirements remain unchanged, but they are rephrased for clarity.
Traceability	Traceability links to other deliverables are changed.
Non-compliance	Requirements are changed, because they do not apply any more to the forthcoming software package. This is the case when the requirements specification is based on that one of a previous project.
Partial compliance	Requirements are changed, because they partially apply to the forthcoming software package. This is the case when the requirements specification is based on that one of a previous project.
Hardware modification	Requirements changes are due to hardware modifications. This type of change usually applies to hardware dependent software requirements.
Add, Delete, Rename variables or parameters	The variables or parameters to which requirements refer change.
Range modification	The range of variables or parameters to which requirements refer changes.

A taxonomy of requirements changes, like the one of Table 3.1, may be used in several ways. It may help to identify requirements issues. For instance, if changes overlap two categories, the affected requirements may need subsequent refinements. This may be the case of too complex or unclear requirements. This is the case when there are requirements that apply both to system software and hardware. The decision whether to allocate requirements to software or hardware may be delayed until further information are available. A side effect of prioritising requirements changes is that usually there exist different evolving paths in terms of requirements changes (e.g., splitting requirements in smaller and more detailed requirements or modify and clarify their specifications). On the other hand constructing a stable taxonomy of requirements change is very difficult. This is because a taxonomy of changes reflects work practice within the specific development environment. For instance, let us assume that an internal policy requires always to delete requirements and to add "new" requirements in stead of modify requirements. The analysis of requirements changes will point out a predominance of added and deleted requirements. Hence any taxonomy of changes should be constructed by taking into account engineering aspects (e.g., specific type of changes) as well as organisational ones (e.g., change policy). Hence on the one hand a taxonomy is a design tool, which may serve to classify as well as monitor requirements changes. On the other hand a taxonomy reflects the work practice within the development organisation.

3.2.3 Requirements Maturity Index

Metrics [Fenton and Pfleeger, 1996] may quantify software properties. The standard IEEE 982 [IEEE, 1988a, IEEE, 1988b] suggests a *Software Maturity Index* to quantify the readiness of a software product. Changes from a previous baseline to the current baseline are an indication of the current product stability. This section applies the Software Maturity Index to software requirements, hence the *Requirements Maturity Index* (*RMI*) quantifies the readiness of software requirements. The *RMI* is an indirect measure that relies on two primitives (or direct measures): R_T and R_C. R_T is the total number of software requirements in the current release. R_C is the number of requirements changes, i.e., added, deleted or modified requirements, allocated to the current release. Equation 3.1 defines the RMI.

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

Figure 3.5 shows the *RMI* calculated for each software release. The *RMI* results sensitive to requirements changes occurring release after release. It fails to capture historical information (e.g., number of releases, cumulative number of changes, average number of changes, elapsed time since introduction of changes, etc.) about requirements changes. Thus every time a substantial subset of requirements changes, the *RMI* decreases because it just reflects the introduction of these changes without capturing any historical evolutionary information. The *RMI* therefore captures stepwise (release-by-release) changes, hence it is too sensitive to changes introduced in a single release over-degrading its assessment about the readiness of requirements.

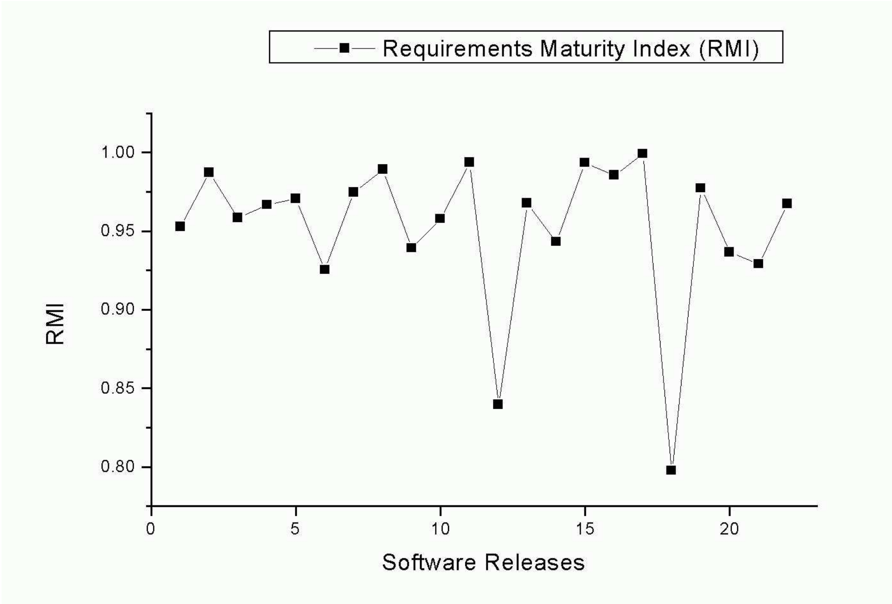


Figure 3.5: Requirements Maturity Index for each software release.

3.2.4 Ageing Requirements Maturity

The behavioural analysis (or performance) of the *RMI* on the case study (see Figure 3.5) points out that the *RMI* lacks to capture the history of changes. It fails to take into account evolutionary information about changes. This is because the *RMI* captures stepwise (release-by-release) changes. Further evolutionary historical information may be obtained by answering questions like: How long have changes been introduced? How many releases were there without changes? How many changes have occurred since the first release? Most of this information can be obtained by indirect measures of requirements features.

We propose to refine the *RMI* by taking into account historical evolutionary information [Anderson and Felici, 2002]. The simplest historical information consists of the *Cumulative Number of Requirements Changes* (CR_C), which measures the number of changes occurred since the first releases. Equation 3.2 based on the CR_C and the number of software releases, n, defines the *Average Number of Requirements Changes* (AR_C).

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

Combining the CR_C and the AR_C together with the RMI, we propose two refinements of the RMI: *Requirements Stability Index* (*RSI*) and *Historical Requirements Maturity Index* ($HRMI$).

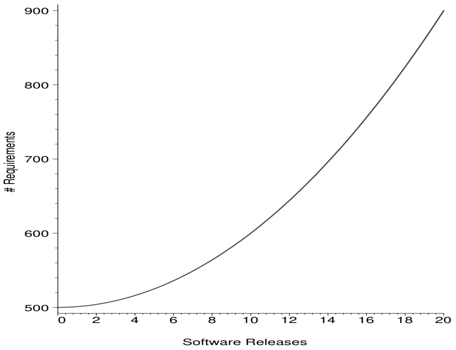
Equation 3.3 defines the *Requirements Stability Index* (*RSI*). The *RSI*, in contrast to the RMI, takes into account the cumulative number of requirements changes, CR_C. Hence *RSI* is sensitive not just to the total number of requirements, R_T, but also to the cumulative number of changes, CR_C. If requirements have never changed since the first releases, the *SRI* assumes its maximum value, 1. This is quite unlikely in practice. The *RSI* will usually assume values below 1 and above 0. But if the cumulative number of changes, CR_C, is greater than the total number of requirements, R_T, the *RSI* will be negative. This is the case of very volatile requirements.

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

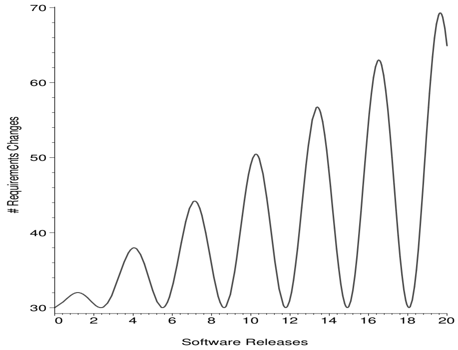
Equation 3.4 defines the *Historical Requirements Maturity Index* ($HRMI$), which relies on the total number of requirements, R_T, and the average number of requirements changes, AR_C. Taking the average number of changes spreads the cumulative number of changes over the number of releases. The result is that the $HRMI$ is smoother that the RMI. Hence the $HRMI$ is les sensitive than the *RMI* to changes over consecutive releases.

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

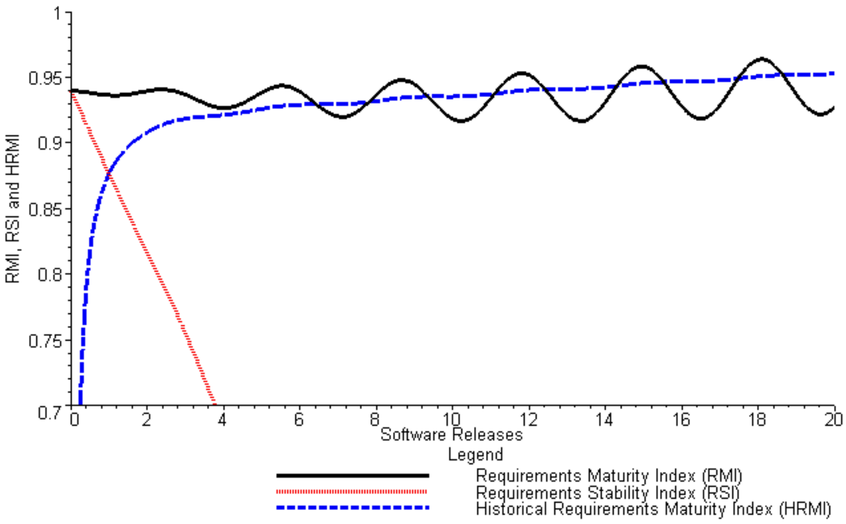
The definitions of RMI, RSI and $HRMI$ are quite similar. All them imply that either the *maturity* or *stability* of requirements is a function of the sizes of requirements and changes. The intuition behind the two refinements, i.e., RSI and $HRMI$, is twofold. Changes affect the stability of the requirements. But the maturity of the requirements depends on the elapsed time (in terms of releases) since the introduction of changes. Hence if requirements stabilise (e.g., the average number of changes decreases), their maturity gradually increases with the delivery of subsequent releases. Figure 3.6 shows the simulation of the above metrics on a sample scenario.



a. Total Number of Requirements.



b. Number of Requirements Changes.



c. The three indexes: RMI, RSI, and $HRMI$.

Figure 3.6: Comparison of the three indexes by simulation on a sample scenario.

Figure 3.6(a) and 3.6(b) respectively show the basic samples of total number of requirements and number of changes. Figure 3.6(c) shows the three indexes: RMI, RSI and $HRMI$. The comparison of the three indexes shows clearly the different behaviours with respect to the sample scenario. The $HRMI$ is less sensitive and more stable than the RMI with respect to changes occurring in consecutive releases.

3.2.5 Ageing Requirements Maturity: Empirical Evidence

This section assesses the proposed quantitative models, i.e., RSI and $HRMI$, by applying them on the case study. Figure 3.7 shows the average number of requirements changes, AR_C, over the software releases. The increasing trend of AR_C characterises the case study. This is probably because many requirements were unclear or roughly defined at the start of the project. Feedback from design, implementation and (integration) testing provides further information to refine the requirements.

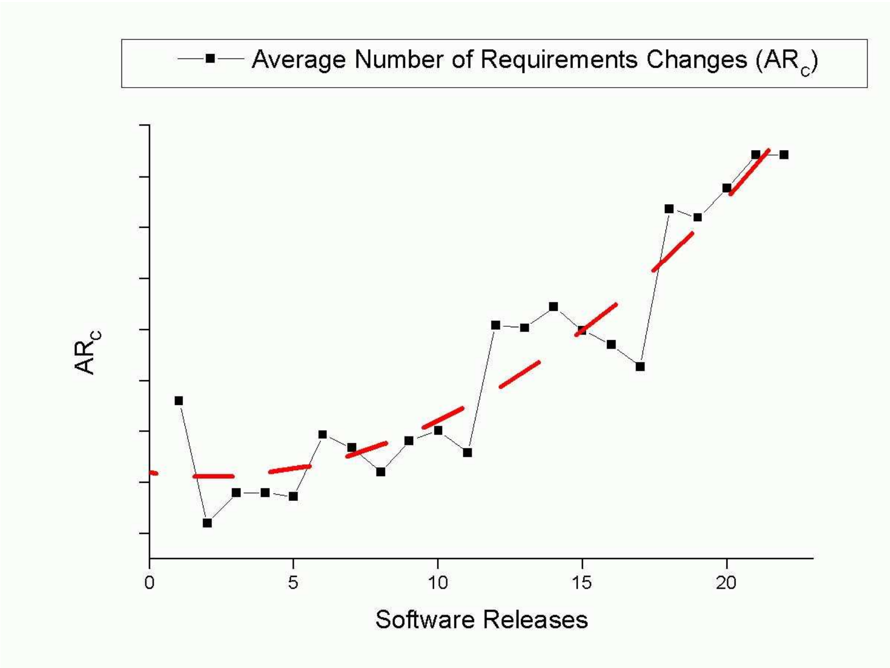


Figure 3.7: Average number of requirements changes (AR_C).

Other case studies may manifest different distributions. Any distribution depends on the specified system, the adopted design process and policy, and the system life cycle.

As shown by the simulation in the previous section, the *RSI* has a decreasing trend. Figure 3.8 shows the *RSI* for each software release. The *RSI* decreases linearly. The *RSI* measures the stability of a set of requirements. A positive *RSI* (i.e., $1 \leq RSI < 0$) means that the Cumulative Number of Requirements Changes (CR_C) is less than the Total Number of Requirements (R_T), hence some requirements were unchanged. A stable set of requirements would have a *RSI* close to 1, whereas a volatile set of requirements would have a *RSI* close to 0 or negative. Similarly, a null or negative *RMI* means that the CR_C equalises or is greater than the R_T. In this unfortunate cases it is quite likely that most of the requirements changed, although there could still be unchanged requirements. This would identify stable requirements in a very volatile set of requirements. As for the AR_C, the linear decreasing trend of the *RSI* characterises the case study. Other case studies may manifest different trends of stabilities.

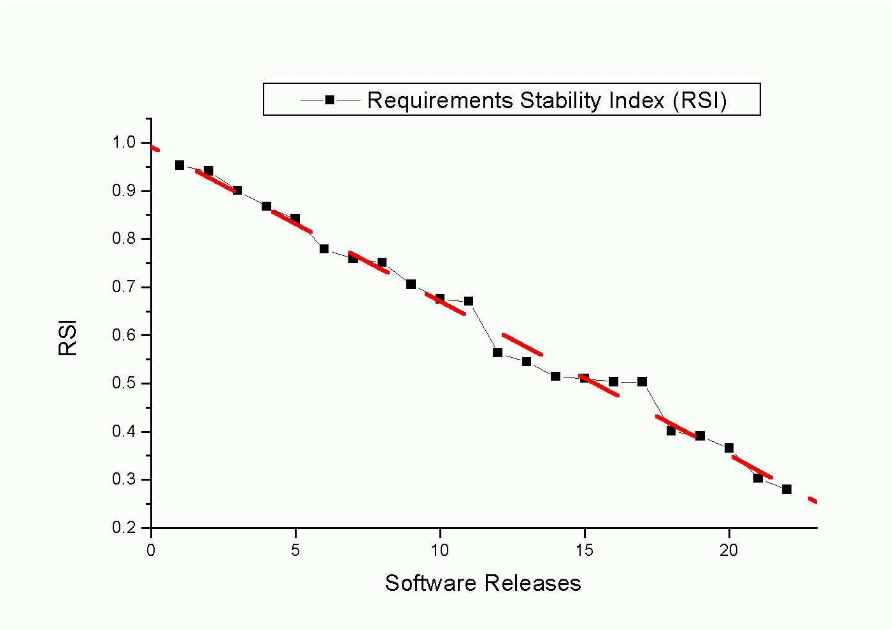


Figure 3.8: Requirements Stability Index (*RSI*).

Figure 3.9 shows the $HRMI$ calculated over the 22 successive releases of the case study. The $HRMI$ smoothly follows the iterations of the development process. That is, decreasing trends are associated with the introduction of major changes (establishing new major releases). The introduction of major changes triggers a stabilisation process over the subsequent releases. During the stabilisation phases the $HRMI$ increases again. $HRMI$ is less sensitive than the RMI to changes over consecutive releases. It captures the requirements process and its stabilisation.

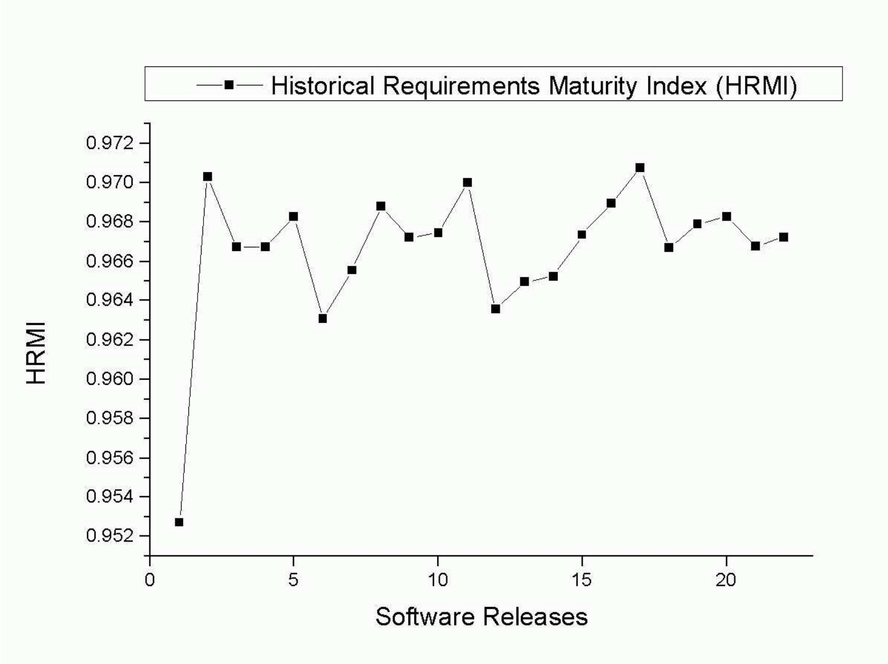


Figure 3.9: Historical Requirements Maturity Index ($HRMI$).

In conclusion this section assesses the proposed metrics, i.e., RSI and $HRMI$. The analysis supports the intuition behind the two metrics. The RSI evaluates the overall stability of a set of requirements. The $HRMI$ evaluates the requirements maturity with respect to the requirements evolution process. The empirical results stress that the distribution of requirements changes is an important driver for the RSI and the $HRMI$. This empirical result relates the requirements evolution process to the development process. The better the development process controls the distribution of changes, the better is the control over the maturity of requirements. Hence the distribution of changes allows us to control the maturity of requirements. Different distributions imply different trends of *RSI* and *HRMI*.

3.2.6 Functional Requirements Evolution

The previous analyses point out the trends of requirements evolution, but they provide us little information about the volatile and stable part of requirements. In order to provide a more detailed analysis our focus [Anderson and Felici, 2002] moves from requirements evolution as a whole towards functional-oriented analyses of requirements evolution. The software functional requirements of the case study fall into eight different functions for which separate documents are maintained. Figure 3.10 shows the number of requirements changes over the software releases for each function, i.e., F1-F8, forming the whole software requirements specification.

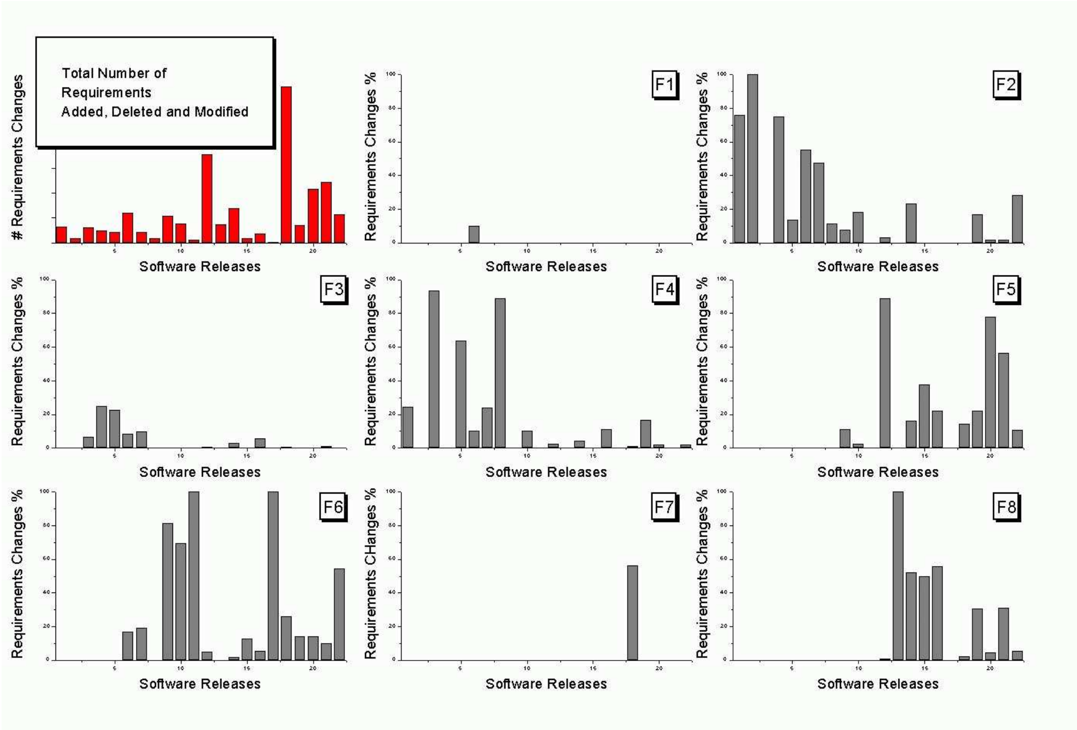


Figure 3.10: Requirements evolution from a functional viewpoint.

The picture in the top-left corner of Figure 3.10 shows the total number of changes for the whole requirements specification. All the other pictures show the percentage of requirements changes occurred in the corresponding release for each function. The functional perspective of Figure 3.10 clearly points out that functions change differently. Some functions are more stable (volatile) than others. For instance, the function F1 changes very little in one of the early releases. F1 therefore is a stable part of the software requirements. The stability of F1 is interesting, because the specific function describes the hardware architecture of the system onto which the software architecture is mapped. Hence the software architecture of the system is stable. This type of information can be useful in future similar projects. The identification of the stable (volatile) parts of requirements is very important in order to establish software product-lines characterised by specific variability points [Bosch, 2000].

The different distributions of changes moreover point out that each function has its own stability, which may change over subsequent software releases. Stable requirements may become volatile, and vice versa. Figure 3.11 shows the cumulative number of requirements changes for each function.

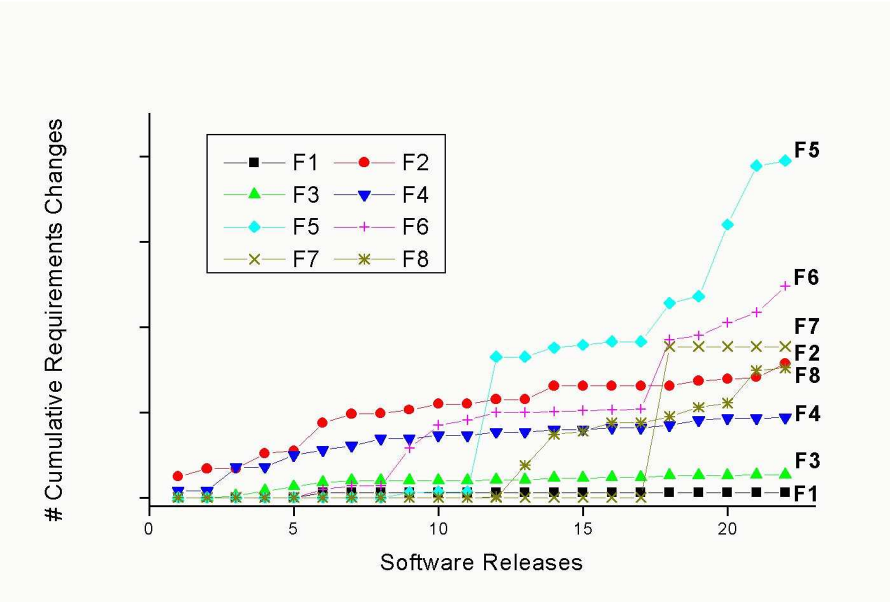


Figure 3.11: Cumulative number of requirements changes for each function.

Figure 3.11 points out that the likelihood that changes can occur into specific functions is inconstant over the software releases. It seems that functions that are likely to change in early software releases change less in late releases, and vice versa. This aspect helps to relate requirements changes to the development process. The different occurrences of requirements changes throughout the development process point out some dependencies among functional requirements [Anderson and Felici, 2001]. Dependencies may be used to effectively assess the impact of changes into requirements and to plan evolution. Understanding these dependencies may improve the requirements process.

The proportion between the number of requirements and the cumulative number of changes for each function provides an intuitive notion of stability. Figure 3.12 shows the scatter plot of requirements size, in terms of number of requirements, versus requirements evolution, in terms of cumulative number of requirements changes at the 22nd software release.

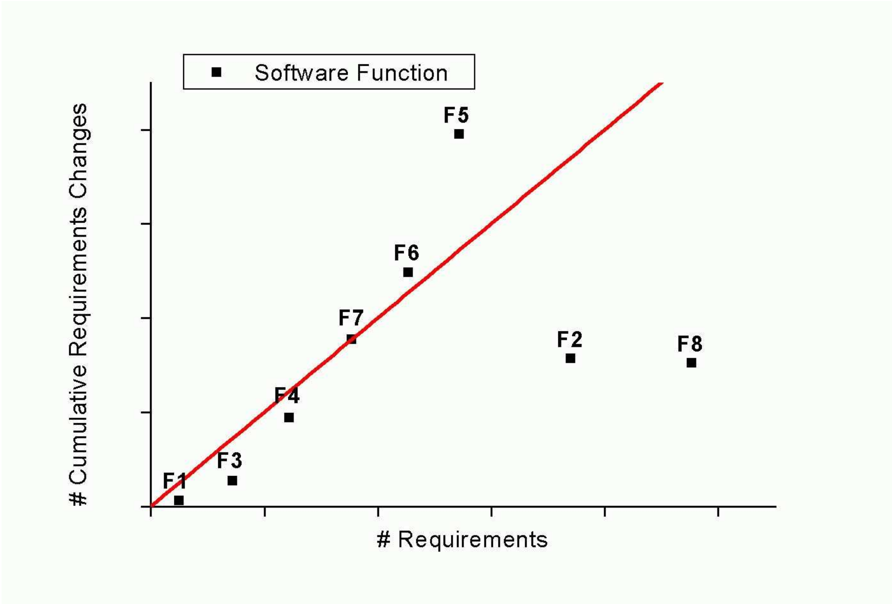


Figure 3.12: Number of requirements versus cumulative number of requirements changes.

The scatter plot investigates the relationship between the cumulative number of requirements changes versus the number of requirements at the 22nd software release.

The aim is to identify whether there is any relationship between sizes of requirements and changes. Intuitively, the functions below the mid line (i.e., the number of requirements equalises the cumulative number of changes) are more stable than the ones above the line. Figure 3.12 shows that there exist a linear relationship between the number of changes occurring into a requirements specification and its size. But there are outliers such as, e.g., F2, F5 and F8. Figure 3.12 provides further confirmation that the function F1, i.e. the software architecture, is a stable part of the system.

The *RSI* similarly provides equivalent stability results. Figure 3.13 shows the *RSI* calculated for each function at the 22nd software release. The functions with negative *RSI*, i.e., F5 and F6, are the most unstable of the system.

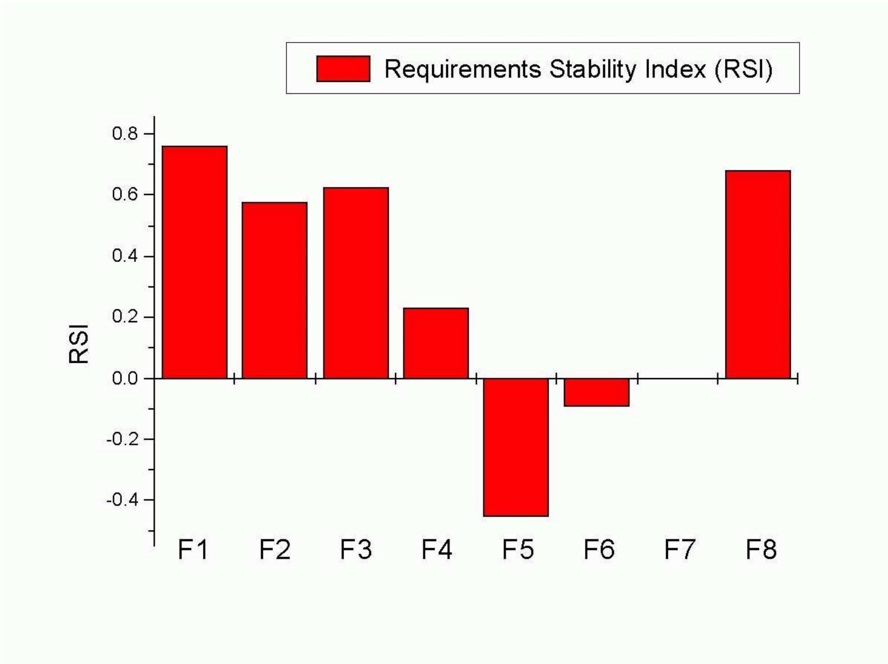


Figure 3.13: Requirements Stability Index for each function at the 22nd software release.

Both figures, i.e., Figure 3.12 and Figure 3.13, rely on the comparison of the sizes of requirements and changes. They provide measurement tools that can be easily applied to any dataset at any stage of the development process.

The functional perspective allows us to acquire further information about requirements evolution. Refinements may provide another way to obtain further understandings. Figure 3.14, for instance, shows the number of requirements changes per software release. Figure 3.14 shows the same requirements evolution trend of Figure 3.3. Figure 3.14 refines Figure 3.3 by taking into account a classification of requirements changes. Requirements changes fall into two major categories: changes that have the authority to modify delivered software, changes that are unauthorised to modify delivered software. The latter consists of changes that affect associated functionalities, but only documentation, traceability or non-delivered code. Implementation may trigger this type of change that usually adapts the documentation to the delivered code.

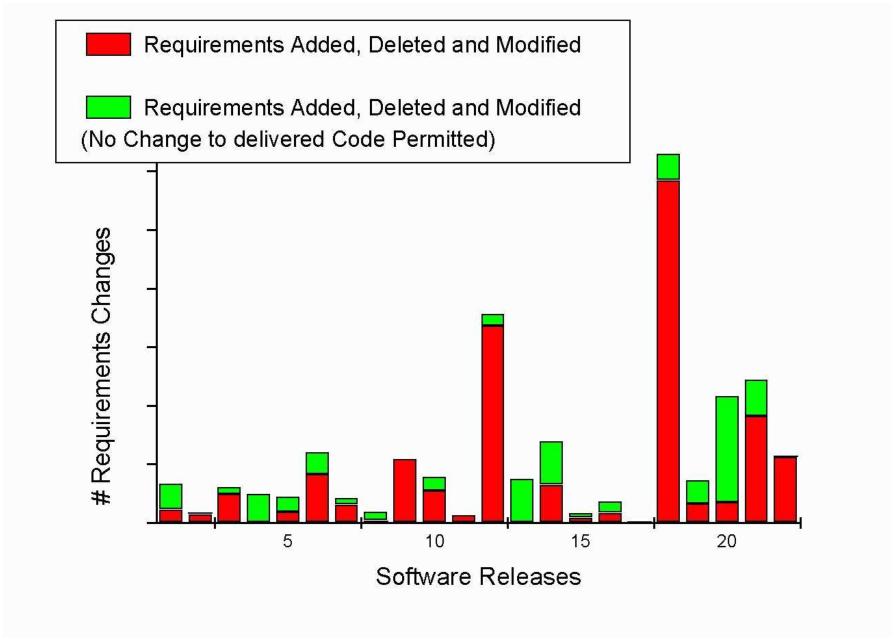
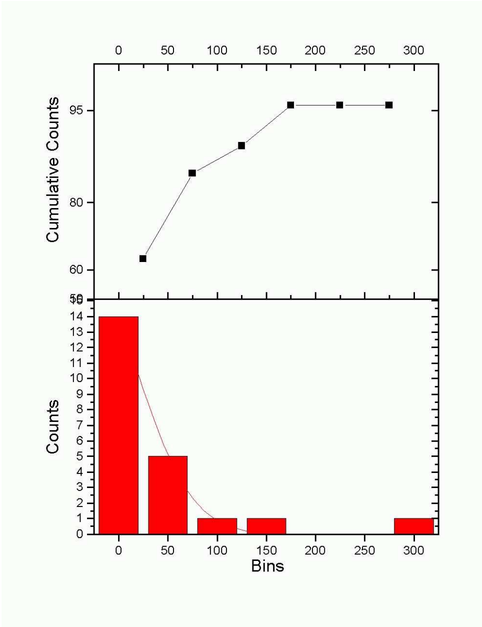
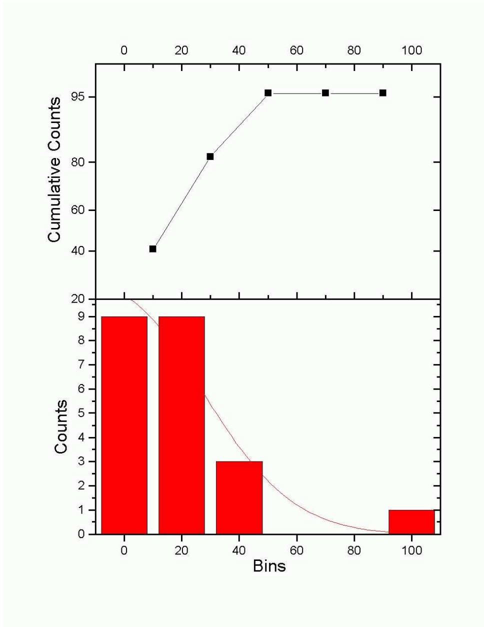


Figure 3.14: Classified requirements changes per software release.

It is furthermore interesting to look at the size of the sets of requirements changes allocated for each software releases. Figure 3.15(a) and Figure 3.15(b) respectively show the histogram for the set of changes affecting delivered code and the one non affecting delivered code. The histograms provide information about the probability of the number of changes allocated to a single release.



a. Changes having permission to modify delivered code.



b. Changes not having permission to modify delivered code.

Figure 3.15: Histograms of the size of the sets of requirements changes allocated to a single software release.

3.2.7 Requirements Dependencies

The functional analysis of requirements evolution points out dependencies between functions. The rationale behind requirements changes may assess the extent to which two functions depend on each other. We evaluate the dependencies between two functions by counting the number of common anomaly reports arose during the software development process. The underlying hypothesis is that the rationale behind requirements changes reflects the dependencies between functions. Thus if requirements changes are due to the same anomaly report, then the affected functions depend on each other. The number of common anomaly reports defines a *Dependency Index* between sets of requirements. Table 3.2 shows the dependencies matrix between the functions forming the software requirements specification. For example, the Dependency Index between the functions F4 and F8 is 5. This means that F4 and F8 have 5 anomaly reports in common. The blank entries in Table 3.2 indicate that the two functions (that identify each blank entry) miss any common anomaly report.

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**

The requirements dependencies matrix provides us a practical tool to assess to which extent software requirements depend on each other. Moreover, it identifies those anomaly reports that trigger changes into different functions. Further analyses of these anomaly reports may point out important information about requirements. The dependencies matrix can be used in several ways. For instance, if similar systems are part of product-lines [Bosch, 2000, Weiss and Lai, 1999], the matrix can monitor dependencies over sequences of successive software products. The dependencies between software functions may decrease over successive software products. Any dependency reduction (e.g., erasing dependencies equal to 1) depends on our ability effectively to identify high modular and low coupling system functions. Thus the dependencies matrix may support the identification of functions forming product-lines within any software organisation.

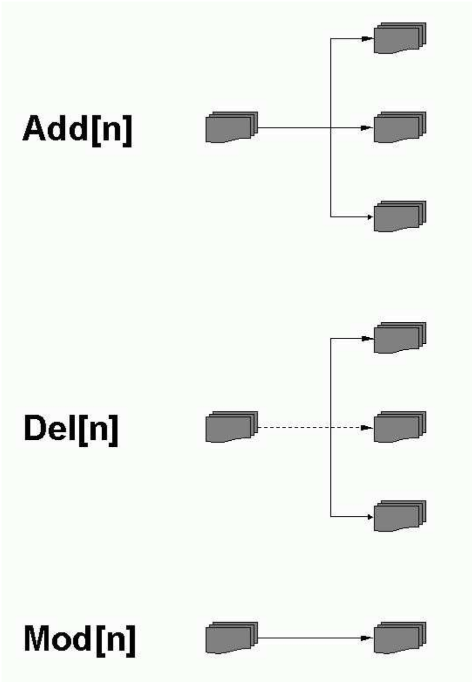
The assessment of the impact of changes relies on requirements traceability. The dependencies matrix may further refine impact analyses. A single change request may trigger a cascade of potential changes. The number of requirements that may potentially be change peaks every time that requests of changes propagate across traceability links. Further information when available will discard most of these latent changes [Hull et al., 2002]. The combination of the dependencies matrix with traceability information may allow us effectively to refine requests of changes by discarding those requests of changes that violate established dependencies. On the other hand the dependencies matrix may highlight those changes that update the dependencies matrix. Despite their usefulness both traceability information and the dependencies matrix fail to capture any temporal dimension of requirements changes. For instance, requirements changes may depend on each other, but they are allocated to different releases. Requirements prioritisation depends on several factors (e.g., costs, stakeholders, development activities, etc.) that constrain the entire development process. Taking into account temporal information about requirements changes may allow us to further refine impact analyses.

3.2.8 Visualising Requirements Evolution

The empirical analyses in the previous sections point out diverse aspects of requirements evolution. This section investigates whether it would be possible to capture some of the evolutionary aspects of requirements by visual modelling. Visual modelling should take account of arising evolutionary aspects of requirements. The visual model should combine a process viewpoint together with an activity one. The aim is to classify, structure and quantify evolutionary features of requirements. Using a graphical model is effective at representing the overall picture of requirements evolution. A diagrammatic representation may furthermore embed more intuitive structures than any text-based representation. The underlying structures should reflect how requirements are engineered throughout the entire development process. Finally a graphical representation may allow us easily to identify similarities over functions and product-lines.

In order to understand work practice we have analysed the requirements specification by inspections [Gilb and Graham, 1993]. The requirements specification consists of eight different documents, one per each software function identified for the system. A unique alphanumerical string identifies each requirement. The format of the alphanumeric string is: *"Document / Section / Requirements id"*. Thus each requirements document represents a collection (or set) of uniquely identified requirements. Each requirements document has therefore a partial order over the alphanumerical id. Browsing through this order relationship, requirements changes have been traced backwards in order to reconstruct the history of the requirements documents. That is, requirements changes in the history of changes have been unfolded backwards. This allows us to analyse the history of the requirements documents throughout the entire development process. The backward reconstruction of the history of changes identifies a structured workflow of requirements evolution. This structure is the basis of our graphical model for requirements evolution.

Our graphical model represents requirements evolution in terms of requirements changes, i.e., added, deleted and modified requirements. For each type of requirement change there is a corresponding representation. Figure 3.16 shows a graphical workflow representation of the three operations[3].



a. Operation workflow.

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.

b. Operation description.

Figure 3.16: Graphical workflow of the three basic requirements changes.

The subset in the middle of the three identifies the subset of changed requirements. In case of modified requirements the set of requirements is unchanged, because the structure of the requirements documents is unchanged by Mod operations. Notice that

[3]Here n represents the size of the subset of changed requirements.

although the graphical representation in Figure 3.16 is informal, the different trees (in terms of different edges and the order of the leafs) imply rough syntax and semantics.

Sequences of requirements changes intuitively capture the requirements evolution process and its complexity. Figure 3.17 shows the requirements evolution workflow for the function F1. The shape (even without detailed information) of the three captures the complexity of the requirement evolution process. The order of the operations is left to right. The most left node (i.e., the root of the three) is the initial set of requirements. The requirements evolution workflow for F1 (see Figure 3.17) is simple and linear. A sequence of basic operations (i.e., Add, Del, Mod) changes the initial set of requirements.

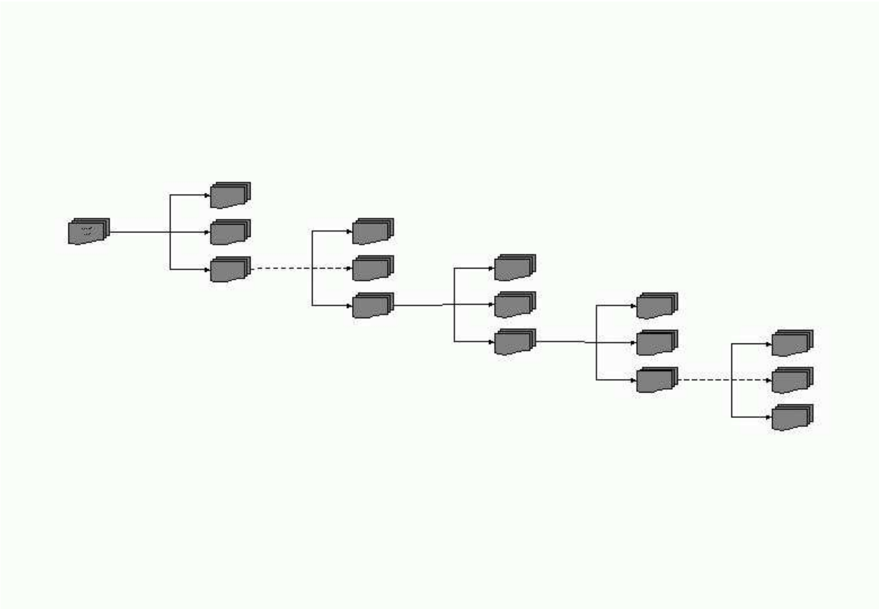


Figure 3.17: Graphical workflow of requirements evolution for F1.

The graphical representation of requirement evolution for the other functions in the case study is more complex than that one of F1. This emphasises how the structured workflow can easily capture the requirements evolution process. Figure 3.18 shows the requirements evolution workflow for the function F4. The requirements evolution workflow of F4 (see Figure 3.18) is more complex than the one of F1 (see Figure 3.17). Requirements changes are allocated over different software releases. This explains the graphical order (left to right) of some operations.

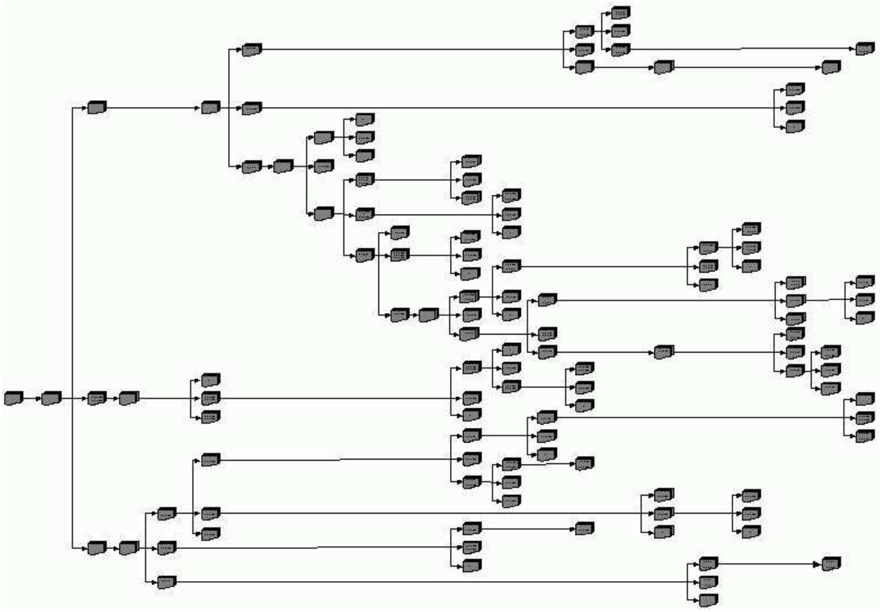


Figure 3.18: Graphical workflow of requirements evolution for F4.

The two workflows show that the graphical representation can capture requirements evolution processes. The graphical representation gives rise to further questions: Is it possible to identify evolutionary processes in terms of requirements changes? Is there any relation between the complexity of requirements evolution workflows and the cost and risk of changing requirements? Is there any relation between requirements evolution paths and system dependability [Felici, 2000, Felici, 2003]? We will further analyse these questions in the remainder of this thesis.

3.2.9 Sequence Analysis

Previous sections point out that it is possible to identify trends and processes of requirements evolution for each function. This section aims to assess whether it is systematically possible to identify requirements evolution processes. Kemerer and Slaughter [Kemerer and Slaughter, 1999] show how *Sequence Analysis* may identify phases in the development process of software systems. They apply Sequence Analysis in order to identify phases of software changes. They show the empirical methodology on two large software systems.

Our goal here is to extend the previous empirical results by Sequence Analysis. The underlying hypothesis is twofold. The basic assumption is that there exist different processes of requirements evolution in terms of requirements change. Moreover, it is possible to identify requirements evolution processes by Sequence Analysis. This analysis differs from the work of Kemerer and Slaughter [Kemerer and Slaughter, 1999] in two main aspects. We are applying Sequence Analysis to requirements changes. We are applying Sequence Analysis in order to identify subprocesses that contribute to the evolution of requirements as a whole.

3.2.9.1 Sequence Analysis: Definition

Sequence Analysis consists of three main steps: Phase Mapping, Gamma Analysis and Gamma Mapping. Table 3.3 describes the three steps that are applied sequentially.

Table 3.3: Description of Phase Mapping, Gamma Analysis and Gamma Mapping.

Analysis	**Description**
Phase Mapping (Step 1)	Phase mapping identifies phases in sequentially ordered events. Phases consist of consecutive occurrence of events of the same type.
Gamma Analysis (Step 2)	Gamma analysis calculates the Goodman-Kruskal gamma score that assesses the proportion of A phases that precede or follow B phases in a sequence. A pairwise gamma score is calculated by $$Gamma(A,B) = \frac{P-Q}{P+Q}$$ where P is the number of A phases preceding B phases, and Q is the number of B phases preceding A phases.
Gamma Mapping (Step 3)	Gamma Mapping orders the phases according to their precedence and separation scores. The precedence score for a phase ranges from -1 to 1 and is the average of its gamma scores. The separation score between a pair of phases ranges from 0 to 1 and consists of the absolute value of the pairwise gamma scores.

3.2.9.2 Sequence Analysis: Empirical Results

The three-step sequence analysis has been performed by WinPhaser[4]. We have used the tool with the default settings (e.g., the minimum phase length is three, that is, three events of the same type occurring in a sequence identify a phase). Requirements changes fall into six types: NEW, MOD, DEL, NDO, MDO and DDO. The first three types[5] (i.e., NEW, MOD and DEL) are requirements changes that affect the requirements and may also affect delivered software. The latest three types of changes[6] (i.e., NDO, MDO, and DDO) are requirements changes that cannot affect delivered software, that is, they can affect just documentation or non-delivered code. Notice that this classification is very general and independent from any domain specific classification (e.g., classification according to the type of change: logical error, compliance, hardware modification, etc.). The remainder of this section shows the results for each step of the Sequence Analysis: Phase Mapping, Gamma Analysis and Gamma Mapping.

Phase Mapping

We first applied Phase Mapping to the entire sequence of requirements changes. Figure 3.19 shows the phase map[7] for all the requirements changes. The map identifies four main phases corresponding to four major releases of the requirements specification. The first phase occurs in the period between 0 and 20. The other phases occur respectively in the periods: 40-50, 70-80 and 90-100. All the phases consist mostly of new requirements. Notice that the map labels some of the phases by *Pending*. Pending phases point out that it is impossible to identify any coherent phase, because the corresponding subsequences consist of collections of different requirements changes. Pending phases may be collapsed with other phases by manual refinements.

As with previous analyses we then perform the Phase Mapping analysis on the

[4]WinPhaser: Sequence Analysis and Comparison, Version 1.1, Copyright ©1989-1995, Michael E. Holmes.

[5]NEW, MODifiy, and DELete requirements.

[6]New (NDO), Modify (MDO) and Delete (DDO) requirements.

[7]All phase maps have been normalised. Thus the scale of the phase length ranges from 0 to 100. The normalisation is useful in order to compare different phases.

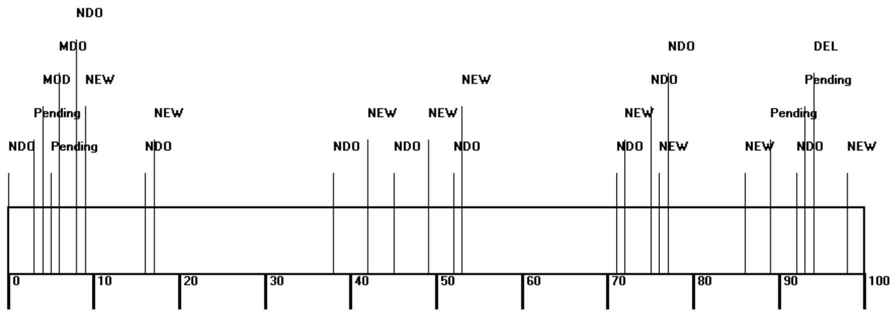


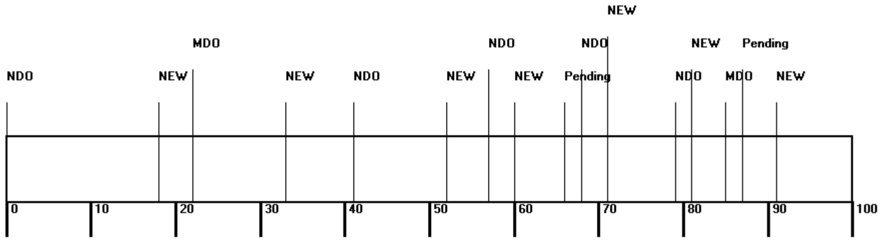
Figure 3.19: Phase map for all requirements changes.

requirements evolution of each software function. Figure 3.20 shows the phase maps for three system functions: F2, F4 and F8.

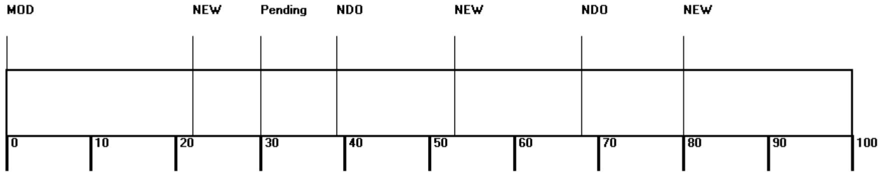
The three maps point out different evolutionary phases for each function. The map for F2, Figure 3.20(a), and the map for F4, Figure 3.20(b), still exhibit Pending phases. These may be due to issues in the requirements process as well as requirements dependencies. The results of Phase Mapping point out different phases of requirements evolution for each function.

Gamma Analysis

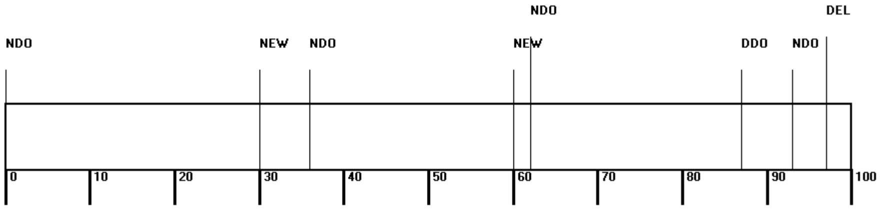
This section shows the Gamma Analysis for three functions: F2, F4 and F8. The Gamma Analysis calculates the gamma score (see Table 3.3) for each pair of types of changes. Two other metrics rely on the gamma score: *precedence* and *separation* scores. The precedence score indicates the order of each phase and can range from -1 to 1. A precedence score that approaches -1 (1) indicates that the phase occurs towards the end (beginning) of the sequence. The separation score quantifies the extent to which two phases are relatively distinct (or overlap) and can range from 0 to 1. Table 3.4, 3.5 and 3.6 respectively show the Gamma Analysis for the function F2, F4 and F8.



a. Phase map for F2.



b. Phase map for F4.



c. Phase map for F8.

Figure 3.20: Phase maps of the requirements changes of three system functions.

Gamma Mapping

Gamma Mapping is the final step of Sequence Analysis. The phases are sequentially ordered from 1 to -1 according to the precedence score. Then phases are grouped together according to their separation scores. A single box identifies the phases with separation scores higher that .50. Phases with separation scores lower that .50 are

Table 3.4: Gamma analysis for F2.

Pairwise Gamma Scores				
	MDO	NDO	NEW	Pending
MDO	.000	.143	-.626	-.905
NDO	-.143	.000	-.622	-.905
NEW	.626	.622	.000	-.418
Pending	.905	.905	.418	.000
Separation Scores				
	MDO	NDO	NEW	Pending
	0.558	0.557	0.555	0.742
Precedence Scores				
	MDO	NDO	NEW	Pending
	0.463	0.557	-0.277	-0.742

Table 3.5: Gamma analysis for F4.

Pairwise Gamma Scores				
	MOD	NDO	NEW	Pending
MOD	.000	-1.000	-1.000	-1.000
NDO	1.000	.000	-.329	1.000
NEW	1.000	.329	.000	.650
Pending	1.000	-1.000	-.650	.000
Separation Scores				
	MOD	NDO	NEW	Pending
	1.000	0.776	0.660	0.883
Precedence Scores				
	MOD	NDO	NEW	Pending
	1.000	-0.557	-0.660	0.217

grouped together into a single box. This is to stress overlaps between phases. The phases with the separation scores in the same range are finally grouped together into a oval box. Notice that this way of grouping phases is slightly different than that adopted by Kemerer and Slaughter [Kemerer and Slaughter, 1999]. We think that this way is

Table 3.6: Gamma analysis for F8.

Pairwise Gamma Scores				
	DDO	DEL	NDO	NEW
DDO	.000	-1.000	.905	1.000
DEL	1.000	.000	1.000	1.000
NDO	-.905	-1.000	.000	.127
NEW	-1.000	-1.000	-.127	.000
Separation Scores				
	DDO	DEL	NDO	NEW
	0.968	1.000	0.677	0.709
Precedence Scores				
	DDO	DEL	NDO	NEW
	-0.302	-1.000	0.593	0.709

suitable in order to group phases that consist of few type of changes. The analysis of Kemerer and Slaughter relies on a richer taxonomy of changes, which implies more phases that overlap each other.

Figure 3.21 shows the gamma maps for the function F2, F4 and F8. The gamma maps clearly show different processes of requirements evolution. In spite of the visual effect of Figure 3.21, the gamma maps fail to capture any temporal relation between (phases of) maps. The gamma map for F8 consists of two major distinct phases. The first one in which new requirements are introduced. The second one in which (incorrect) requirements are deleted. While F8 evolves according to a two-phase evolution process, F2 and F4 evolve according to different processes. They both have "Pending" phases, but occurring in different periods (towards the end for F5 and towards the beginning for F4). They do not have clear periods in which requirements are deleted. Despite this there might be deleted requirements in the Pending phases. The Pending phases may be due to requirements dependencies or issues in the requirement process. This may provide useful information in order to focus the analysis of requirements processes. Furthermore, the phases for F2 appear to be closely related each other. This may be due to the fact that F2 evolves through most of the software releases.

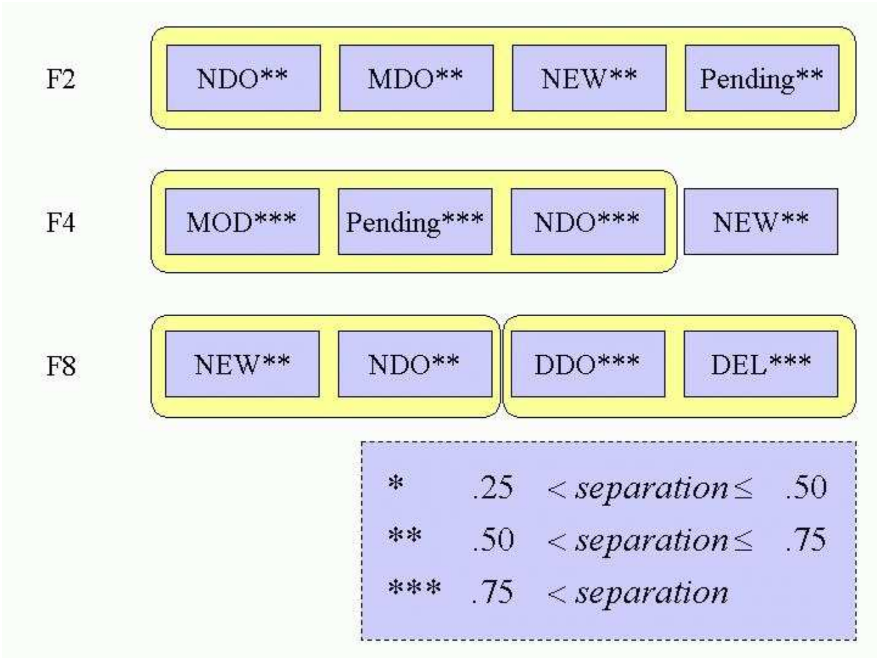


Figure 3.21: The gamma maps for three functions.

3.3 Lessons Learned

The case study enhances our understanding of requirements evolution drawn from industry. In summary, our experience is twofold. On the one hand we faced the practical problem of collecting evolutionary information. On the other hand we analysed the evolutionary features of requirements. These two main points are discussed in what follows.

3.3.1 Requirements Evolution Practice

The case study drawn from industry provides us some practical challenges. Similar challenges may arise in other industrial contexts. Behind any challenge there is actually an issue to deal with. The remainder of this section discusses the main practical challenges: *Data Collection*, *Data Organisations and Goals*, and *Enhanced Visibility*.

Data Collection. Building a data repository of evolutionary data is a difficult task. There are various critical aspects that affect data collections under evolutionary scenarios. The collection of data should be well integrated into the development process. Poorly integrated data collection will result in increased workload and frustration for people who are supposed to collect data. Moreover, people will drop any data collection activity under the pressure of forthcoming deadlines. This will result in out of date data repositories. Substantial effort will be required in order to update these repositories during final stages of the development process. In the worst case the repositories will become unusable and ineffective. They will moreover fail to provide any evolutionary feedback in the development process.

Data Organisations and Goals. The organisation of evolutionary data is another aspect concerning an effective collection of evolutionary data. Data organisation affects our ability to analyse and identify evolutionary features. Unsuitable organisation will provide limited support to identify any emergent information. Data organisations should fulfil specific goals and address specific issues. Why are data collected? Is any data analysis foreseen? What are the expected outcomes? Who will review/read/use any (emergent) information? Answering these questions will help to organise an effective data repository. For instance, assume that a simple history of changes is the main record of requirements changes. The history of requirements changes easily provides evidence of tracking changes for certification purpose. But it fails to provide any feedback in the development process. This is because it lacks any support to identify evolutionary relationships.

Enhanced Visibility. Issues relevant to requirements evolution affect project visibility. Poor coordination between different organisational layers may reduce the overall visibility within the project. Moreover, it affects our ability to assess the impact of changes (or to perform any sensitivity analysis). A trade-off between process and product management may tackle visibility issues. On the one hand process management is useful to standardise system developments, but it limits our visibility over software products. On the other hand product management enhances product features in development processes, but it affects process repeatability across different systems.

3.3.2 Requirements Evolution Features

This section summarises the outcomes of the empirical analysis of the case study with respect to requirements evolution. The requirements evolution features emphasise the specific case study, but it is possible to identify similar features in other case studies across different industrial contexts. The generality of the empirical investigation allows us easily to replicate the analyses. We summarise the main requirements evolution features in what follows.

Quantitative Requirements Evolution. The measurement of requirements evolution requires a well-defined standard policy to classify requirements changes. Even a simple classification of requirements changes implies specific work practice and policy. For instance, the three-type classification of Added, Deleted and Modified requirements identifies the activities of: adding new requirements into the requirements specification, deleting requirements from the requirements specification and modifying requirements in the requirements specification. A management policy based on traceability information should support requirements management with respect to types of changes. For instance, requirements should be uniquely identified by an alphanumeric identification, which furthermore allows us uniquely to link changes and requirements[8]. The combination of type of change together with traceability information allows us to measure various aspects of requirements evolution. Moreover, the combination of traceability and type of change easily supports the analysis of the impact of changes. Although the impact of changes is critical for project management, it needs subsequent refinements that take into account various requirements aspects (e.g., changes criticality, type of change, history of changes, etc.).

Taxonomy of Requirements Changes. The investigation of the history of changes points out a taxonomy of requirements changes. The resulting taxonomy characterises the specific case study as well as its industrial context. The identification of a taxonomy of requirements changes supports the introduction of standard work practice in the

[8] Sommerville and Sawyer describe various guidelines for requirements engineering practice [Sommerville and Sawyer, 1997a].

development environment. On the one hand a taxonomy will help to reduce biases due to different experiences and expertise. Moreover, a taxonomy will support the monitoring of requirements evolution. On the other hand the identification of a stable taxonomy may require experience across several projects.

Ageing Requirements Maturity. Any simple account of requirements maturity may be misleading. Our experience shows that any estimation (e.g., by the Requirements Maturity Index) of requirements maturity should be carefully evaluated in the specific context. Any measurement of requirements maturity should be interpreted against the specific development process (e.g., data collection activities, certification constraints, changes classifications, etc.) and management policy (e.g., priorisation of requirements changes, certification constraints, allocation of requirements changes, etc.). Requirements maturity should furthermore take account of ageing factors for requirements. For instance, the elapsed time since requirements were introduced (or modified) may be useful to refine any assessment of requirements maturity. This type of refinement allows us better to link the requirements process with the development process. On the other hand the concept of *maturity* can be misleading and misunderstanding for requirements. Future research should further investigate how to distinguish diverse requirements maturities. Further investigations should address the relation between stability, volatility and maturity.

Functional Requirements Evolution. The empirical investigation of the case study points out that it is possible to identify change-prone requirements from a functional viewpoint. The analysis identifies stable and volatile requirements. Moreover it emphasises different distributions of requirements changes for each function. This information may be useful in order to identify reusable requirements (e.g., stable requirements of the system architecture) as well as to devise product-lines ranging around specific variability points. The different distributions of requirements changes point out dependencies between functional requirements. Requirements dependencies may be useful to refine the assessment of the impact of changes. On the other hand requirements dependencies may be further refined through subsequent releases and similar projects (within the same product-line).

Requirements Evolution Processes. The case study points out our inability to visualise requirements evolution. Our preliminary attempts of visualising requirements evolution stresses that any visual representation should take into account evolutionary process features (e.g., releases, activities, etc.) as well as product features (e.g., type of requirements changes). The representation of requirements evolution processes may allow us to identify similarities between processes and to distinguish them with respect to their complexity. The visualisation of requirements evolution and the identification (by sequence analysis) of different requirements processes show that it is possible to identify different requirements process for each function. This provides us new insights to investigate requirements evolution in future research.

Chapter 4

A Smart Card Case Study

This chapter describes a viewpoint analysis of a case study drawn from the smart card industry. The analysis relies on interviews and questionnaires. In spite of sparse data, the analysis points out many issues that characterise live production environments. Although changes affect several viewpoints and increase project risk, they are part of learning and understanding processes in software production. From the analysis it is evident how even a single change affects many different socio-technical aspects.

4.1 Description of the Case Study

Smart card systems are ubiquitous in our 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*, *Interactive* and *Security* systems.

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 requests of any provided service depend on almost random human factors. Operational profiles show that human-computer interaction is a critical factor 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 smart card context we consider here is certified according to many quality and security standards. Among these its management process complies with PRINCE2, a process-based project management approach integrating a product-based project planning [CCTA, 1998]. PRINCE[1] (PRojects IN Controlled Environments) is a structured method for project management. It is used extensively by the UK Government and is widely recognised and used in the private sector, both in the UK and internationally. The investigation of the smart card context identifies general aspects of managing requirements. Although we had access to sparse data, the analysis points out many issues that characterise live production environments. The remainder of this chapter describes a viewpoint analysis of the smart card case study. The analysis relies on interviews and questionnaires. On one hand viewpoint discrepancies trigger changes and increase project risk. On the other hand viewpoint discrepancies provide information needed for control [Weinberg, 1997]. Discrepancies are part of learning and understanding processes in software production. From the analysis it is evident how even a single change affects many different socio-technical aspects.

4.2 Empirical Investigation

Our analysis of the smart card context consists of stakeholder interviews in the production environment. The interviews focus on requirements engineering practice (e.g., requirements management policy) as well as product features (e.g., requirements change, system boundaries, etc.). A *Requirements Engineering Questionnaire* (see Appendix

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

A) allows us to articulate and structure stakeholder interviews. The questionnaire consists of 152 questions grouped in three sets that address three different viewpoints: *Business*, *Process* and *Product*. In addition we inspected [Gilb and Graham, 1993] a range of project documents (e.g., Change Request Form, Issue Log, Change Log Progress, etc.) to allow us to identify practical examples that help to interpret the responses to the questionnaire. In this section we provide an analysis of a smart card production environment. The analysis investigates different requirements viewpoints [Sommerville and Sawyer, 1997a]. Each viewpoint corresponds to different stakeholders (or management levels [Weinberg, 1997]). Each viewpoint has a preferred type of requirement that is their main area of concern [PROTEUS, 1996]. Although viewpoint discrepancies can trigger requirements changes, viewpoints provide different understandings of requirements changes, hence requirements evolution. The viewpoints analysis furthermore points out both process and product aspects.

4.2.1 Requirements Evolution Viewpoints

The analysis of the smart card organisation highlights three different hierarchical viewpoints named *Business*, *Process* and *Product* viewpoints. They identify different management levels and responsibilities within the organisation. Each level corresponds to different processes and requirements. The three viewpoints provide a hierarchical management structure that deals with requirements changes. All the three viewpoints together contribute to the overall management of requirements changes.

The interviews point out how different viewpoints seek different management support. The business viewpoint would like to increase project visibility[2]. Although the smart card context complies with several process standards, project managers would like to improve their visibility on the product. On the other hand many engineers (e.g., programmers, designers, etc.) seem to dislike any control over their work. They hesitate whenever management (or anyone else) tries to exercise some control over the product of their work. Thus, although the business viewpoint retains control over the

[2] "From what we know about controlling a project, and from what we know about the natural imperfections of human beings, the concept of private ownership of work products cannot be part of any process model. Instead, every process model must conform to the Visibility Principle: Everything in the project must be visible at all times." [Weinberg, 1997], p. 226.

overall process, it can easily loose visibility over the product. This limits any ability to modify the product by tuning the process.

Although process-oriented methodologies allow to plan project activities, they usually provide limited support to tailor processes to product features. This often requires a shift from process to product-oriented software management. The process viewpoint would like to enhance its management ability by measuring requirements evolution. Although the management process keeps track of requirements changes, software metrics simply capture ongoing trends. The process viewpoint would like to improve its measurement practice by taking into account product as well as environmental aspects.

The Product viewpoint would like to enhance its ability to identify reusable software functions. It moreover would like to define repeatable processes to allocate functions to smart card system requirements. At this level there are two opposing processes. The first one (top-down) splits and refines requirements. This creates an information flow expansion throughout the development process. The second one (bottom-up) allocates specific functions to software requirements. This second process could be improved and enhanced by taking into account past experience. Figure 4.1 shows a schematic representation of the gap between the two opposing processes.

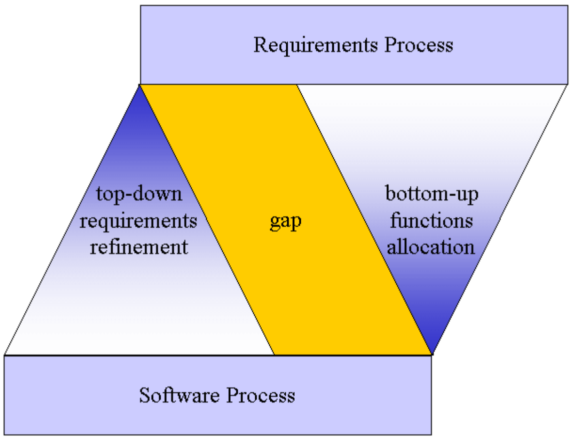


Figure 4.1: A schematic representation of the gap between the two opposing processes existing at the product level.

The gap between these two processes represents the extent to which an organisation is able to identify an optimal and effective set of (reusable) software functions. The smaller the gap, the better the ability in reusing software functions and identifying product-line features[3]. Although each viewpoint deals with different production aspects, all viewpoints would like differently to enhance their ability to deal with requirements changes. The remainder of this section describes the change management features of each viewpoint.

Business Viewpoint. It identifies a high management level within the organisation. The business management is responsible for starting new smart card projects. At this organisational level business stakeholders interact in order to define system requirements. There are three major business stakeholders: the business core, the customer and the bureau. The business core consists of those stakeholders (e.g., project managers, marketing managers, etc.) who are responsible for the spin-off of business cases (i.e., production of smart card systems) related to the supply of smart card systems. The customer consists of those stakeholders who acquire smart card systems in order to support their own business (e.g., credit card suppliers, banks, pay-tv providers, etc.). The bureau consists of those stakeholders (e.g., software engineers, testing engineers, process managers, production line managers, etc.) who are responsible for the production of commissioned smart card systems. Figure 4.2 shows how these stakeholders interact each other by a high-level requirements process.

The business core provides a portfolio of (generalised) smart card systems. Each smart card system consists of a product-line that relies on specific technological artifacts (e.g., JAVA card, card management, public key infrastructure interfaces, etc.) that support system functionalities (e.g., digital signature, electronic identity, mobile commerce, etc.). The customer provides further system requirements that constrain the new smart card systems. Thus, the smart card system requirements consist of general requirements completed by customer requirements. The bureau department is

[3]"Requirements ideas often emerge from the system development process and flow upward into the requirements process. A good requirements process authorizes the participation of developers, focuses their efforts and make them visible, yet keep them under some sort of control. Without this focus, developers have difficulty making the very real contribution only they can make." [Weinberg, 1997], p. 275.

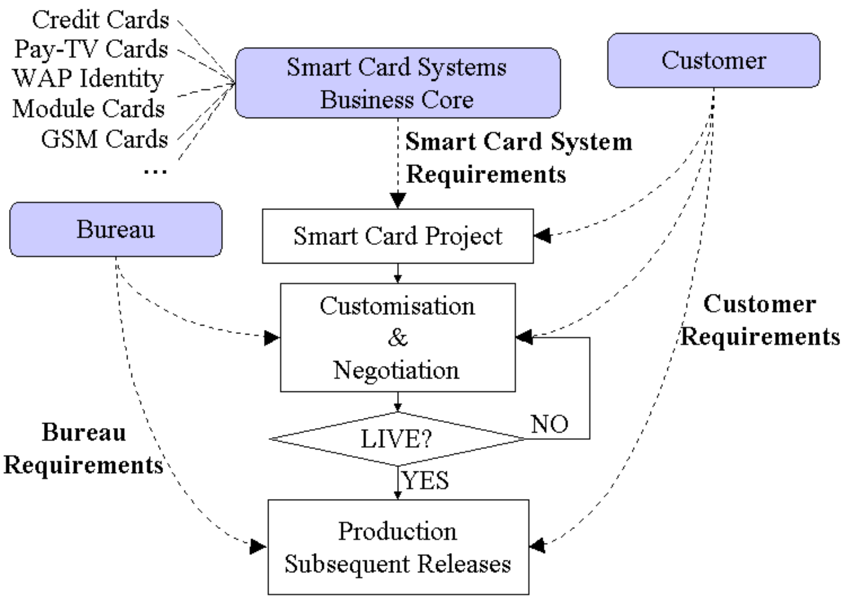


Figure 4.2: The high-level smart card requirements process.

responsible for the production of smart card system projects. Notice that the production involves the physical production of the smart cards (i.e., hardware production) as well as the software programming of the smart cards (i.e., software production). The bureau further constrains (e.g., by additional production requirements) and negotiates these requirements. When the business stakeholders agree on a requirements baseline, the project is declared LIVE and the smart card production begins. The production of subsequent deliveries involves several management and development processes (e.g., change management process, software development process, etc.) tailored for the production of the commissioned smart cards. During the production of each delivery new requirements may arise due to smart card system usage (e.g., misbehaviours), further customer requirements (e.g., new smart card services, additional cards, etc.) or business constraints (e.g., production issues).

The business viewpoint therefore deals with the management of system and business requirements. These are indirectly related to software requirements. The high-level requirements process takes into account any relevant arising issue. A list of classified[4] issues is maintained throughout any smart card project. Table 4.1 shows some examples of issues. It is evident that issues involve many socio-technical aspects (e.g., work practice, organisational responsibilities, hardware, software, etc.) of smart card systems.

Table 4.1: Examples of issues that required changes.

Issue Type	**Description**
Change	*The current version of the Intranet site does not suit Customer Services' way of working, there are also issues with printing and viewing through different browsers.*
Change	*Who will be responsible for supporting the system once it has gone live?*
Change	*Support of the Linux Server:* *1. Only 2 members of staff are experienced with Linux and they are not on call, what happens if a problem occurs when they are unavailable?* *2. Can the system sustain downtime from Friday evening until Monday morning?* *3. What is the impact of major hardware failures as it could take 2-3 days to rectify?* *4. Is spare hardware in place / required?*
Change	*X does not currently have visibility of batches processed by the new Z system.*
Change	*X does not currently store Operator details.*
General	*There is a conflict with the software if both X and Y are run simultaneously on a PC, this results in an error. Action: The decision has been made to accept this issue and to write additional exception handling into the software to trap the error.*

Most of these issues are concerned with high-level system requirements and may trigger changes. Both the bureau and the customer report issues encountered with the commissioned smart cards. Thus the issues can be relevant to the smart cards as well as to their production. A unique request of change corresponds to each issue classified as Change. All requests of changes require further investigation in order to establish

[4]Types of issues are, e.g., Risk, General, Concern, Change.

what modifications are needed and to which extent. The change management process takes into account all the change requests. Although the Change issues may trigger changes, other issues (e.g., General) may also trigger changes that are easily accepted and accommodated. The other types of issues (e.g., Risk) are usually concerns that may eventually trigger changes. But any definitive action (e.g., change) has been delayed until further information are available. Any type of issue is monitored from the business viewpoint, then the process viewpoint deals and manages all needed changes.

Process Viewpoint. It consists of all the management processes adopted within the organisation [CCTA, 1998]. The process management level takes into account all change requests. Changes requests are due to issues encountered in the smart card production as well as usage. Change requests can be traced backwards to the bureau as well as to the customer. The process management maintains a list of all request of changes and the related actions. Table 4.2 shows examples extracted from a list of changes together with the relevant needed actions. Note that the associated actions can imply software changes as well as procedural or organisational changes.

Table 4.2: Examples of change progress reports.

Change Description	**Action**
A new interface to X is required to allow visibility of batches downloaded in the new Z system.	*A Software Request has been issued. This change is to be handled as a second phase to the project as it falls outside of the initial scope. It should not have any detrimental effect on the system delivery date.*
Operator details to be stored in the MySQL at point of progress scan.	*This change is to be handled as part of phase 2, it should not have any detrimental effect on the system delivery date.*
24x7 IT support for X IT systems.	*A defined procedure is required within IT to log and progress support calls out of normal working hours, a detailing procedure on Linux sever support required.*

Figure 4.3 shows the change management process that deals with all requests for change. The initial part of the change management process is a macro-process of the negotiation activity. If changes require some software development the process starts a set of subsystem analyses. Each analysis corresponds to a different subsystem of the whole smart card system. The analyses assess the impact of change on each subsystem. The impact of change estimates the cost of changes in terms of man-power (e.g., man-day). The set of impact reports serves as basis for the negotiation of changes. The negotiation activity will produce a software change request for each agreed change.

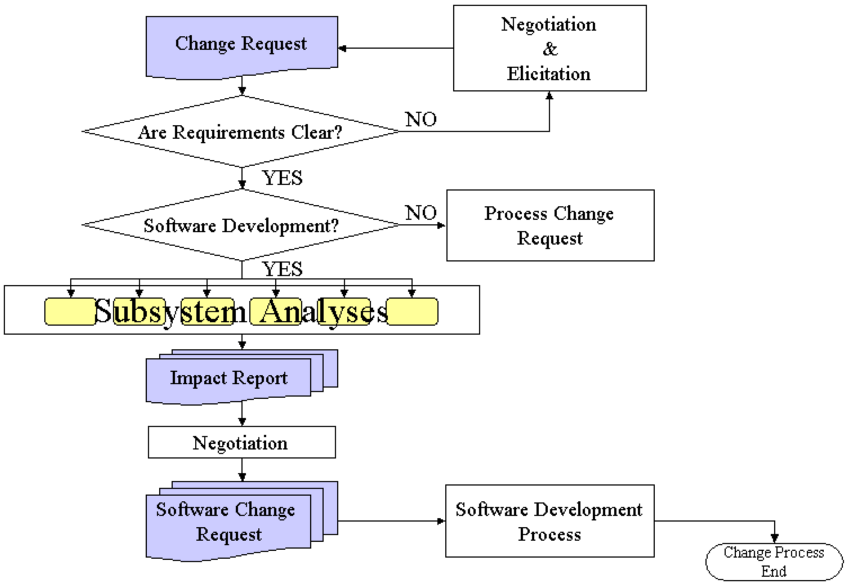


Figure 4.3: The change management process.

Figure 4.4 shows an example extracted from a change request form. The change request form identifies the change description, the suggested modification and the change rationale. The software development process finally takes into account the software requests of changes. The software development process corresponds to a software product viewpoint.

Product Viewpoint. This identifies the software production level in the smart card organisation. The software management level deals with all software requests of

Change Description: *The operator identification should be saved when asked for in the scanning applications.*
Suggestion for Modification: *Recording Operator Identification - the following 2 tables should be added to the MySQL database to record which operator updated a batch status.*
⋮
Change Rationale: *Operator ID will be a useful piece of data for historical analysis by bureau management.*

Figure 4.4: An example extracted from a change request form.

changes. Figure 4.5 shows an example extracted from a software change request form.

Required Functionality: *A new interface needs to be added to X to allow visibility batches downloaded in the new Z system. The interface will work similarly to the one between B and X. ... The current development and implementation of X should be unaffected. An additional development and implementation phase should deliver the new interface by early December, to coincide with the launch of Z.*
Testing Scenarios:

- *Check to ensure that interface software picks up jobs from new Z and correctly populates the MySQL tables and new Z tables.*

⋮

- *The new functionality should be fully tested and accepted on the test server before being migrated onto the main X server.*

Figure 4.5: An example extracted from a software change request form.

The software development process takes into account all software changes. Figure 4.6 shows the V model tailored to the smart card organisation. Each software request of change triggers the entire development process (i.e., from Requirements Analysis to Production Integration Maintenance). The development process elaborates the software requirements with respect to the change requests. It moreover allocates requirements changes to subsequent releases of the smart cards. That is, requirements and software changes are allocated and prioritised according to the production release schedule of the smart cards. Differently from the process viewpoint that estimates changes in terms of man-power [Brooks, 1995], the product viewpoint estimates the cost of software changes in terms of development activities (e.g., coding and testing). Notice that the different type of change estimation may cause discrepancies and disagreements between the business viewpoint (e.g., project managers) and the product viewpoint (e.g., software engineers)[5].

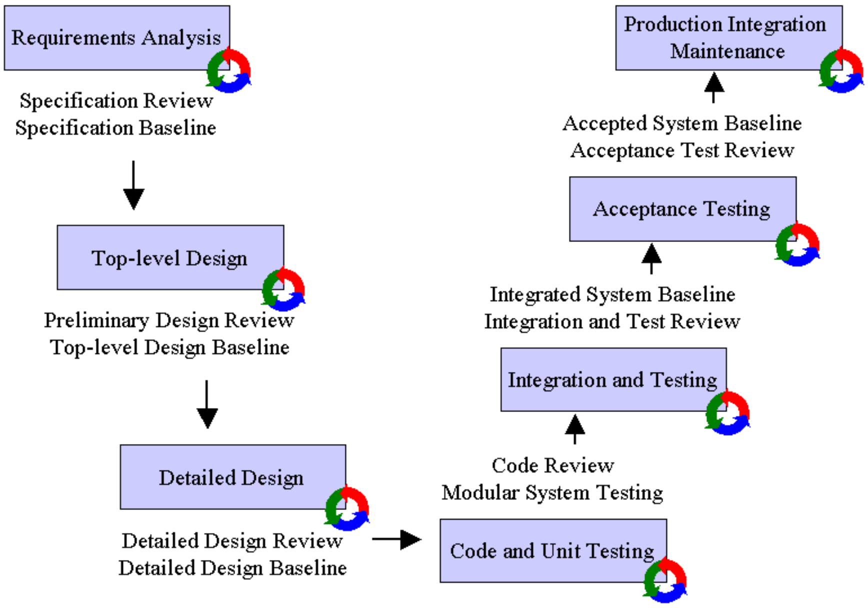


Figure 4.6: The V model adopted by the smart card organisation.

[5]"The second fallacious thought mode is expressed in the very unit of effort used in estimating and scheduling: the man-month. Cost does indeed vary as the product of the number of men and the number of months. Progress does not. *Hence the man-month as a unit for measuring the size of a job is a dangerous and deceptive myth.* It implies that men and months are interchangeable." [Brooks, 1995], p. 16.

4.2.2 Viewpoint Analysis

A requirements engineering questionnaire (Appendix A) allows us to structure the interviews. The questionnaire assesses the general understanding of the requirements engineering practice within an organisation. The questionnaire consists of 152 questions organised into three main sets, i.e., Business, Process and Product requirements engineering. Figure 4.7 shows the 18 different groups of questions[6] that form the questionnaire.

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

Figure 4.7: The groups of requirements engineering questions.

Figure 4.8 shows the (average) profiles[7] of three persons with similar experience and with different responsibilities within the smart card organisation. The three persons, who filled in the questionnaire, correspond to different management levels within the organisation. The two project managers, associated to the process viewpoint, are responsible for the change management processes and resources. The software development manager, associated to the product viewpoint, is responsible for the entire software development process. The questionnaire captures how people perceive the requirements engineering practice within the organisation. Figure 4.8 clearly shows that

[6]Each question allows a multiple-choice answer: *N/A*, Not Applicable, if the person is unconcerned with the addressed question; *UN*, Unknown, if the person is unable to answer according to her/his knowledge, skills or position within the organisation; the other answers consist of five different levels, namely, *VL* (Very Low), *L* (Low), *A* Average, *H* (High) and *VH* (Very High). If the person is able to judge according to her/his knowledge, skills or position within the organisation. Note that the questionnaire only subjectively assesses the requirements engineering practice within the organisation.

[7]Each answer corresponds to the following numerical values: 0 (N/A), 0 (UN), 1 (VL), 2 (L), 3 (A), 4 (H) and 5 (VH).

the software development manager has a lower confidence than the project managers on the requirements engineering practice within the organisation. On one hand management processes easily deal with high-level smart card requirements. Thus, project managers trust the requirements engineering practice. On the other hand management processes provide limited support for software requirements. Therefore, the software development manager struggles with requirements engineering practice.

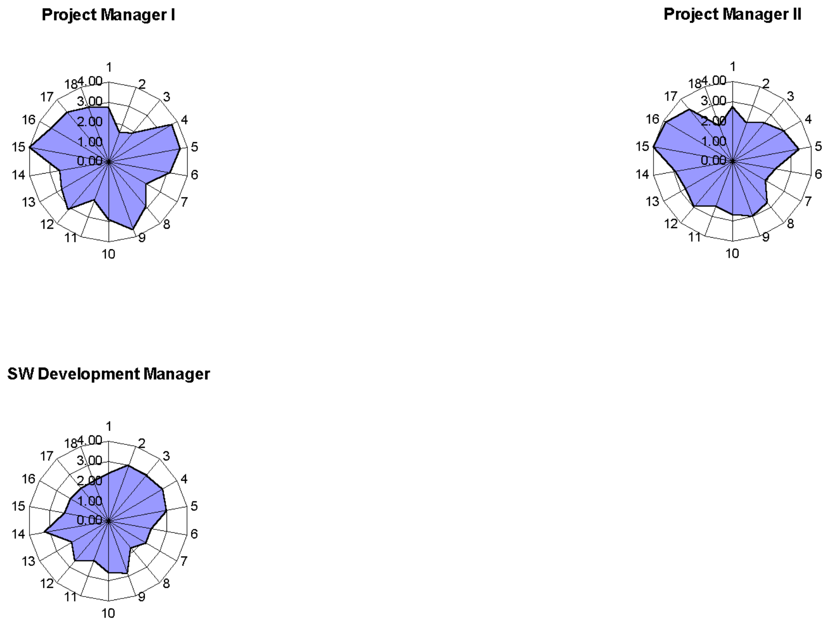


Figure 4.8: Three profiles captured by the requirements engineering questionnaire.

Figure 4.9 shows a different representation of the three profiles. This representation is convenient to identify major divergences. It is interesting to notice that the two similar viewpoints (i.e., the viewpoint associated to the project managers) have similar trends. Whereas the third viewpoint diverges in particular points. The largest divergences correspond to the groups of questions 2, 3, 8, 15, 16 and 17. The group 2 and 3 identify business aspects of the requirements engineering practice. Interestingly, they also disagree on the group 8, which consists of questions related to requirements evolution and maintenance. This is probably due to the process-oriented management of requirements changes. Although it could be due to communication issues within the organisation. Management hierarchies[8] have properly to communicate requirements changes through each level in the organisation [Reason, 1997, Weinberg, 1997]. On the other hand requirements engineering practice can also provide limited support to communicate specific types of requirements changes (e.g., requirements changes that affect software).

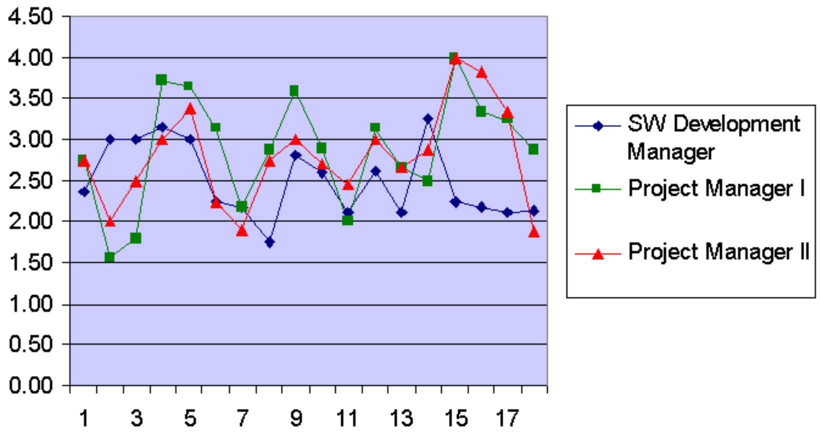


Figure 4.9: A different representation of the three profiles.

Finally, the other major divergencies correspond to the group 15, 16 and 17 (see Figure 4.9), which identify product-oriented questions. These point out different understandings of smart card systems. The project managers, who deal with high-level system requirements, easily identify system boundaries. Whereas, the software development manager exhibits a low confidence in the system interface. This is due to

[8] "Software engineering management is concerned with choices about technology in a broad sense of that term, and those choices are made at many levels. If control is not in place at one level, it becomes more difficult to control at another. For a software engineering organization to be well managed, *all* levels have to be well managed. But, in addition, they have to be coordinated with one another. Therefore, software engineering managers have to decide on what level to place various control responsibilities. We know that decisions at the higher levels act to control decisions as lower levels, but decisions at low levels can also control decisions at the higher levels. We need to be aware of why we put which control decision on which level." [Weinberg, 1997], pp. 190-191.

the fact that low-level software interfaces often interact with other (external) systems. Software is a subtle part of smart card systems. Project managers have often a limited visibility of the software issues. All these divergences identify concerns for the requirements engineering practice.

4.3 Lessons Learned

The viewpoint analysis highlights issues in the requirement engineering practice drawn from the smart card case study. In spite of sparse data, the analysis effectively points out many requirements evolution aspects that characterise live software production environments. Although changes affect several viewpoints (or management levels) and increase project risk, they are part of learning and understanding processes in software production. From the analysis it is evident how even a single change affects many different socio-technical aspects.

Requirements Evolution Viewpoints. The analysis identifies three different hierarchical viewpoints named *Business*, *Process* and *Product* viewpoints. Each viewpoint corresponds to different processes and requirements within the organisation. For instance, management processes easily deal with high-level system requirements, although they provide limited support for low-level software requirements. This points out struggles with requirements engineering practice. Although viewpoint discrepancies often cause requirements issues (e.g., inconsistent, incorrect, etc.), all viewpoints provide as a whole a hierarchical management structure that deals with requirements changes. Interestingly, each viewpoint differently perceives requirements evolution. Thus, on one hand viewpoint interactions (hence, stakeholder interactions) give rise to requirements evolution. On the other hand viewpoint interactions represent a mechanism to capture and take into account requirements changes.

Viewpoint Management Support. Each viewpoint seeks different management support. Although process-oriented methodologies allow the planning of project activities, they usually provide limited support to tailor processes to product features. This often requires a shift from process to product-oriented software management. On one hand management processes keep track of requirements changes. On the other hand quantitative approaches (e.g., software metrics) should take into account product as well as environmental aspects. This allows us to identify reusable (product-line) functions. It moreover would be possible to define repeatable processes to allocate low-level software functions to high-level system requirements. There are usually two opposing processes. The first one (top-down) splits and refines requirements. This creates an information flow expansion throughout the development process. The second one (bottom-up) allocates (according to past experience) specific low-level software functions to high-level system requirements. The gap between these two processes represents the extent to which an organisation is able to identify an optimal and effective set of (reusable) software functions. The smaller the gap, the better the ability in reusing low-level software functions and identifying high-level system requirements. Although each viewpoint deals with different requirements, all viewpoints seek support dealing with requirements change.

Requirements Issues. The structured interviews, using the requirements engineering questionnaire, effectively highlight common requirements issues in live software production environment. Viewpoint divergences clearly point out the different understanding of the requirements engineering practice within the organisation. Unsurprisingly, requirements evolution finds little agreement among the different viewpoints. Although process-oriented management properly capture requirements changes, it provides limited support to development activities. Moreover hierarchical organisations often struggle to communicate requirement changes through each management level (or viewpoint). On the other hand requirements engineering practice provides limited support to communicate specific types of requirements changes. Another major issue is the identification of system boundaries. Although a holistic viewpoint captures many aspects of socio-technical systems, viewpoints provide different understandings of (software) systems. Each viewpoint captures different system boundaries. For instance, a project management level easily identifies high-level system requirements, although these under-specify low-level software interfaces. Software, as a subtle part of socio-technical systems, has limited visibility from high-level holistic viewpoints. All these issues highlight concerns for the requirements engineering practice.

Chapter 5

Modelling Requirements Evolution

This chapter introduces a formal framework to model and capture requirements evolution. The framework relies on a heterogeneous account of requirements. Heterogeneous engineering provides a comprehensive account of system requirements. Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings [Bergman et al., 2002a, Bergman et al., 2002b]. The formal extension of solution space transformation defines a framework to model and capture requirements evolution. The resulting framework is sufficient to interpret requirements changes. The formal framework captures how requirements evolve through subsequent releases. Hence, it is possible to define requirements evolution in terms of sequential solution space transformations. Intuitively, requirements evolution identifies a path that browses solution spaces.

5.1 Heterogeneous Requirements Engineering

Research and practice in requirements engineering highlight critical software issues. Among these issues requirements evolution affects many aspects of software production. In spite of the increasing interest in requirements issues most methodologies provide limited support to capture and understand requirements evolution. Unfortunately, the underlying hypotheses are often unable to capture requirements evolution [Weinberg, 1997]. Although requirements serve as basis for system production, development activities (e.g., system design, testing, deployment, etc.) and system usage feed back system requirements. Thus system production as a whole consists of cycles of discoveries and exploitations. The different development processes (e.g., V model, Spiral model, etc.) diversely capture these discover-exploitation cycles, although development processes constrain any exploratory approach that investigates requirements evolution. Thus requirements engineering methodologies mainly support strategies that consider requirements changes from a management viewpoint. In contrast, requirements changes are emerging behaviours of combinations of development processes, products and organisational aspects.

Heterogeneous engineering considers system production as a whole. It provides a comprehensive account that stresses a holistic viewpoint, which allows us to understand the underlying mechanisms of evolution of socio-technical systems. Heterogeneous engineering[1] involves both the system approach [Hughes and Hughes, 2000] as well as the social shaping of technology [MacKenzie and Wajcman, 1999]. On one hand system engineering devises systems in terms of components and structures. On the other hand engineering processes involve social interactions that shape socio-technical systems. Hence, stakeholder interactions shape socio-technical systems. Heterogeneous engineering is therefore convenient further to understand requirements processes. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings [Bergman et al., 2002a, Bergman et al., 2002b].

This chapter describes a formal extension of solution space transformation, hence heterogeneous requirements engineering, in order to capture requirements evolution. The underlying hypothesis is that heterogenous requirements engineering is sufficient to capture requirements evolution. The formal extension of solution space transfor-

[1] "People had to be engineered, too - persuaded to suspend their doubts, induced to provide resources, trained and motivated to play their parts in a production process unprecedented in its demands. Successfully inventing the technology, turned out to be heterogeneous engineering, the engineering of the social as well as the physical world.", [MacKenzie, 1990], p. 28.

mation defines a framework to model and capture requirements evolution. The resulting framework is sufficient to capture and model requirements evolution. The formal framework captures how requirements evolve through subsequent releases. Hence, it is possible to define requirements evolution in terms of sequential solution space transformations. Intuitively, requirements evolution identifies a path that traverses solution spaces. The remainder of this chapter introduces the formal extension of solution space transformation.

5.2 Heterogeneous Requirements Modelling

Requirements engineering commonly considers requirements as goals to be discovered and (design) solutions as separate technical elements. Hence requirements engineering is reduced to be an activity where technical solutions are documented for given goals or problems. Heterogeneous engineering [Bijker et al., 1989] further explains the complex socio-technical interactions that occur during system production. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings [Bergman et al., 2002a, Bergman et al., 2002b].

Requirements therefore are socially shaped (that is, constructed and negotiated) [MacKenzie and Wajcman, 1999] through sequences of mappings between solution spaces and problem spaces [Bergman et al., 2002a, Bergman et al., 2002b]. These mappings identify a *Functional Ecology* model that defines requirements as emerging from solution space transformations. The Functional Ecology model describes solution-problem iterations of the form:

$$solution \rightarrow problem \rightarrow solution\ .$$

This implies that requirements engineering processes consist of solutions searching for problems, rather than the other way around (that is, problems searching for solutions). Figure 5.1 shows how the *Solution Space Transformation*[2] unfolds. A solution space

[2] $\mathcal{GS}$, Global Solution Space; $\mathcal{S}$, Local Solution Space; $\mathcal{S}_t$, Current Solution Space; $\mathcal{AS}$, Anomaly Space; $\mathcal{P}$, Problem Space; $\mathcal{P}_t$, Proposed Problem Space; $\mathcal{S}'$, Future Local Solution Space; $\mathcal{S}_{t+1}$, Proposed Solution Space; $\mathbf{R}_o^t$, objective or functional requirements; $\mathbf{R}_c^t$, constraining or non functional requirements.

(i.e., a current solution space $\mathcal{S}_t$) solves some highlighted problems (i.e., proposed system problem space $\mathcal{P}_t$). The contextualisation of the selected problems into the initial solution space identifies the system requirements (i.e., objective or functional requirements $\mathbf{R}^t_o$) as mappings between solution and problem spaces. The resolution of these problems identifies a future solution (i.e., a proposed solution space $\mathcal{S}_{t+1}$). The mappings between the solved problems and the future solution define further system requirements (i.e., constraining or non functional requirements $\mathbf{R}^t_c$). This heterogeneous account of requirements is convenient to capture requirements evolution.

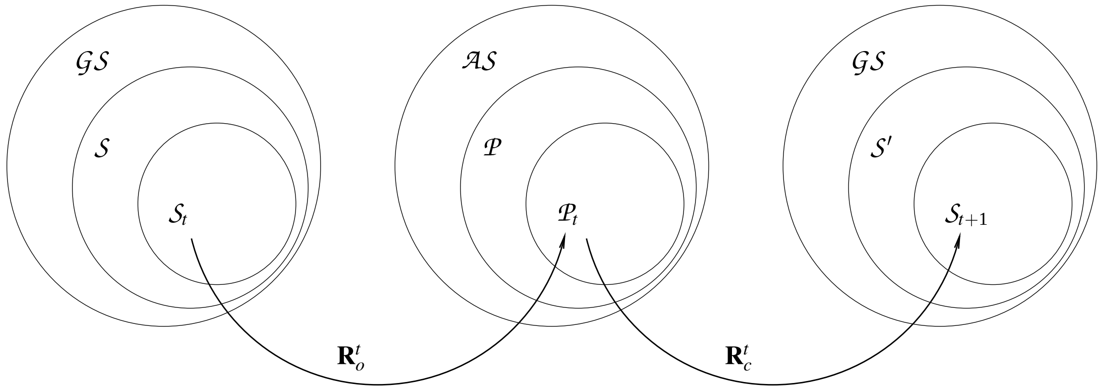


Figure 5.1: The solution space transformation.

This section introduces a formal extension of the solution space transformation. The basic idea is to provide a formal representation of solutions and problems. The aim of a formal representation is twofold. On one hand the formalisation of solutions and problems supports model-driven development. On the other hand it allows us to formally capture the solution space transformation, hence requirements evolution. The formalisation represents solutions and problems in terms of modal logic[3].

Intuitively, a solution space is just a collection of solutions, which represent the organisational knowledge acquired by the social shaping of technical systems. Solutions

[3]Appendix B briefly introduces propositional modal logic [Chagrov and Zakharyaschev, 1997, Fitting and Mendelsohn, 1998]. Propositional modal logic provides enough expressiveness in order to formalise aspects of the solution space transformation. The formal extension of the solution space transformation relies on logic bases: syntax, semantics and proof systems. All definitions can be naturally extended in terms of other logics (e.g., [Stirling, 2001]) bases (i.e., syntax, semantics and proof systems). The definitions still remain sound and valid due to construction arguments.

therefore are *accessible possibilities* or *possible worlds* in solution spaces available in the production environment. This intentionally recalls the notion of possible world underlying *Kripke models*. Thus, solutions are Kripke models. Whereas problems are formulas of (propositional) modal logic. Collection of problems (i.e., problem spaces) are issues (or believed so) arising during system production. Kripke models (i.e., solutions) provide the semantics in order to interpret the validity of (propositional) modalities (i.e., problems). Note that it is possible to adopt different semantics of the accessibility relations in Kripke models. For instance, the accessibility relation can capture design information like state transitions. Therefore, the accessibility between two possible worlds or system states would mean that it is possible to access one state from another one through the accessibility relation. This semantics is used throughout this chapter to show the development of a design of a simple clock, as explanatory example. Another semantics of the accessibility allows the gathering of evolutionary information about requirements (i.e., evolutionary requirements dependency) in the next chapter. This accessibility relation identifies dependencies between (set of) requirements (or functional requirements). This semantics captures evolutionary information at an higher level of granularity (or abstraction) than design. Using different semantics for interpreting the accessibility relation highlights and captures diverse evolutionary information. Although the examples use two different semantics (i.e., temporal state transition and evolutionary dependency), it is beyond the scope of this work to decide which semantics should be used in any specific case. On the one hand, the use of different semantics highlights the flexibility of the given framework. On the other hand, it requires careful considerations when used in practise to capture diverse evolutionary aspects of requirements. Based on the syntax of Kripke models, proof systems (e.g., *Tableau systems*) consist of procedural rules[4] (i.e., inference rules) that allow us to prove formulas validity or to find counterexamples (or countermodels). The remainder of this section introduces the formal definitions that extend the solution space transformation.

[4] Note that there exist different logics (e.g., **K**, **D**, **K4**, etc.) that correspond to different proof systems. Appendix B briefly introduces different logics and the corresponding proof systems and inference rules. Any specific proof system implies particular features to the models that can be proved. The examples use the different logics as convenient to explaining. It is beyond the scope of this work to decide which proof system should be used in any specific case.

5.2.1 Solution Space

Technical solutions represent organisational knowledge that may be available or unavailable at a particular time according to environmental constraints. A *Local Solution Space* is the collection of available (or believed available) solutions in an organisation.

Definition 5.1 (Local Solution Space) *A Local Solution Space, $\mathcal{S}$, is the current solution space and all locally accessible solution spaces that can be reached from the current solution space using available skills and resources offered by the principals (or business stakeholders).*

The definition of Local Solution Space relies on the notion of reachability between solution spaces. The notion of reachability (between solution spaces) is similar to the notion of *accessibility* in Kripke structures. In spite of this similarity, the use of Kripke structures as underlying models was initially discarded due to organisational learning [Bergman et al., 2002a]. Although Kripke structures fail to capture organisational learning, they can model solutions. Each solution therefore consists of a Kripke model (or Kripke structure or frame) within the proposed formal framework. Thus, a Local Solution Space is a collection of Kripke models, i.e., solutions. A sequence of solution space transformations then captures organisational learning. Although solution spaces depend on several volatile environmental constraints (e.g., budget, human skills, technical resources, etc.), solution space transformation captures organisational learning by subsequent transformations. Hence, a sequence of solution space transformation captures organisational learning. Hence, requirements evolution (in terms of sequences of solution space transformations) is a process of organisational learning. The feasibility of solution spaces identifies a hierarchy of solution spaces.

Definition 5.2 (Global Solution Space) *A Global Solution Space, $\mathcal{GS}$, is the space of all feasible solution spaces, including those not currently accessible from the Local Solution Space. The feasible solution spaces require mobilisation of all principals and technologies to be accomplished. All local solution spaces exist within relevant global solution spaces. That is, $\mathcal{S}$ is a subspace of $\mathcal{GS}$.*

Feasible solutions are those available within an organisation (e.g., previous similar projects) or that can be reached by committing further resources (e.g., technology outsource or investment). In terms of Kripke models, a Global Solution Space, is the space of all possible Kripke models. Some of these models represent solutions that are available (if principals commit enough resources) within an organisation. Whereas, others would be unavailable or unaccessible. Finally, the notion of *Current Solution Space* captures the specific situation of an organisation at a particular stage.

Definition 5.3 (Current Solution Space) *The Current Solution Space, denoted as $\mathcal{S}_t$, embodies the history of solved social, technical, economic and procedural problems (i.e., socio-technical problems) that constitute the legacy of previously solved organisational problems at current time t. The Current Solution Space exists within a Local Solution Space. That is, $\mathcal{S}_t$ is a subspace of $\mathcal{S}$.*

The Current Solution Space therefore captures the knowledge acquired by organisational learning (i.e., the previously solved organisational problems). In other words, the Current Solution Space consists of the adopted solutions due to organisational learning. This definition further supports the assumption that solution space transformations capture organisational learning, hence requirements evolution. It is moreover possible to model the Current Solution Space in terms of Kripke models. $\mathcal{S}_t$ is a collection of Kripke models. Let briefly recall the notion of Kripke model. A Kripke model, $\mathcal{M}$, consists of a collection G of *possible worlds*, an *accessibility relation* R on possible worlds and a mapping $\Vdash$ between possible worlds and propositional letters. The $\Vdash$ relation defines which propositional letters are true at which possible worlds. Thus, $\mathcal{S}_t$ is a collection of countable elements of the form

$$\mathcal{M}_i^t = \langle G_i^t, R_i^t, \Vdash_i^t \rangle \ . \tag{5.1}$$

Each Kripke model then represents an available solution. Thus, a Kripke model is a system of worlds in which each world has some (possibly empty) set of alternatives. The accessibility relation (or alternativeness relation), denoted by R, so that $\Gamma R \Delta$ means that Δ is an alternative (or possible) world for Γ. For every world Γ, an atomic proposition is either true or false in it and the truth-values of compound non-modal propositions are determined by the usual truth-tables. A modal proposition $\Box\varphi$ is regarded to be true in a world Γ, if φ is true in all the worlds accessible from Γ. Whereas, $\Diamond\varphi$ is true in Γ, if φ is true at least in one world Δ such that $\Gamma R \Delta$. In general, many solutions may solve a given problem. The resolution of various problems, hence the acquisition of further knowledge, narrows the solution space by refining the available solutions. An example clarifies all the given definitions throughout this chapter. The example is the design of a simple clock.

Example 5.1 (A Clock) *A Kripke model easily captures a (design) solution for a simple clock. Figure 5.2 shows a graphical representation of the Kripke model* $\mathcal{M}_i^t$ *(i.e., clock design solution).*

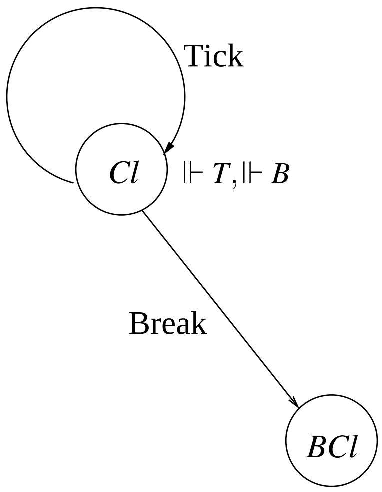


Figure 5.2: A Kripke model for a clock.

Let $\mathcal{M}_i^t = \langle G_i^t, R_i^t, \Vdash_i^t \rangle$ *be defined as follows. The set of possible worlds is* $G = \{Cl, BCl\}$*, where Cl stands for "Clock" and BCl stands for "Broken Clock". As any normal clock, the world Clock can reach itself by ticking, that is, by the accessibility relation* $Cl\,R_i^t\,Cl$*. Otherwise if it breaks, it can reach the world Broken Clock by the reachability relation* $Cl\,R_i^t\,BCl$*. The truth assignments* $Cl \Vdash T$ *and* $Cl \Vdash B$ *take into account the different accessibility relations, which respectively mean that Clock can tick or can break. Of course, there are many other (design) solutions that model a ticking system. The Current Solution Space simply contains all available solutions at time* t*.*

The further development of the clock example shows how the (formally augmented) solution space transformation captures the evolutionary requirements process that identifies the solution solving the arising problems.

5.2.2 Problem Space

The Functional Ecology model defines the role of requirements with respect to solutions and problems. Requirements are mappings between solutions and problems, as opposed to being solutions to problems. Problems then assume an important position in order to define requirements. Likewise the case studies, any observation is initially an anomaly. According to environmental constraints (e.g., business goals, budget constraints, technical problems, etc.) stakeholders then highlight some anomalies as problems to be addressed. On one hand problems identify specific requirements with respect to solutions. On the other hand any shift in stakeholder knowledge causes problem changes, hence requirements changes. The anomaly prioritisation identifies an hierarchy of problem spaces.

Definition 5.4 (Anomaly) *An anomaly is an inconsistency, observed by some stakeholder, between the current solution space and a desired solution space believed by the stakeholder to be achievable within the current solution space.*

Definition 5.5 (Anomaly Space) *The Anomaly Space, named $\mathcal{AS}$, is the set of all anomalies associated with the system under consideration.*

An anomaly[5] identifies the assumptions under which the system under consideration should work. Thus, anomalies represent concerns that stakeholders may regard as system problems to be solved eventually.

Definition 5.6 (Problem) *A Problem is an anomaly that is observed and acted upon by a principal and thereby placed into organisational agenda of need-to-solve anomalies.*

Definition 5.7 (Problem Space) *A Problem Space, $\mathcal{P}$, is the space of all problems implied by a current solution space, $\mathcal{S}_t$, by all its principals. A problem space (i.e., the space of all selected problems) is by definition always a subspace of an anomaly space.*

Definition 5.8 (Proposed System Problem Space) *A Proposed System Problem Space, $\mathcal{P}_t$, is the space that contains all the recognised problems chosen by the principals at*

[5] "The observation of an *anomaly* or *presumptive anomaly* provides the assumptions (derived from science) under which indicate either that under some future conditions the conventional system will fail (or function badly) or that a radical different system will do a much better job.", [MacKenzie, 1990], p. 69.

time t that justify the proposed system. P_j^t *is a problem (*1 *of at least m problems) within* $\mathcal{P}_t$.

The formal representation of anomalies and problems has to comply with two main requirements. Firstly, it has to capture our assumptions about the system under consideration. Secondly, it has to capture the future conditions under which the system should work. Modalities[6] provide a logic representation of problems (or anomalies). Note that the *possible worlds* model (which underlies the modal logic semantics by Kripke structures) is the core of well-established logic frameworks for reasoning about knowledge [Fagin et al., 2003] and uncertainty [Halpern, 2003].

Example 5.2 *Propositional Modal Logic allows us to express modalities. Figure 5.3 shows the Modal Square of Opposition [Fitting and Mendelsohn, 1998] due to Aristotle in his De Interpretatione.*

It is necessary that *P*		It is necessary that not *P*
It is not possible that not *P*		It is not possible that *P*
	$\nwarrow \;\; \nearrow$	
	$\swarrow \;\; \searrow$	
It is not necessary that not *P*		It is not necessary that *P*
It is possible that *P*		It is possible that not *P*

Figure 5.3: The Modal Square of Opposition.

Formulas of propositional modal logic capture system properties like Safety and Liveness. For instance, let consider the formula $\Box P \rightarrow P$*. The formula means that if a property P is valid at every accessible possible worlds, then it is actually valid at the real world. It represents a simple safety property that states "nothing bad ever happens".*

[6] "A modal qualifies the truth of a judgement. Necessarily and possibly are the most important and best known modal quantifiers. They are called *alethic* modalities, from the Greek word of *truth*. In traditional terminology, *Necessarily P* is an *apodeictic judgement*, *Possibly P* a *problematic judgement*, and *P*, by itself, an *assertoric judgement*. The most widely discussed modals apart from the alethic modalities are the temporal modalities, which deal with past, present, and future...", [Fitting and Mendelsohn, 1998], p. 2.

Another example is the formula $\Box P \rightarrow \Diamond P$. The formula means that if the property P is valid at every accessible possible worlds, then it will be valid eventually. It represents a simple liveness property that states "something good eventually happens".

Modalities therefore capture problems highlighted by stakeholder and allow us to reasoning on solutions. That is, the logic representation of solutions (in terms of Kripke models) and problems (in terms of modalities) allows us to assess whether solutions address selected problems (i.e., fulfil selected properties). Moreover, the logic framework captures mappings between solutions and problems, hence requirements [Bergman et al., 2002a, Bergman et al., 2002b]. As opposed to solutions pointing to problems, *Problem Contextualisation* is the mapping of problems to solutions.

5.2.3 Problem Contextualisation

The stakeholder selection of a Proposed System Problem Space, $\mathcal{P}_t$, implies specific mappings from the Current Solution Space, $\mathcal{S}_t$. *Problem Contextualisation* is the process of mappings problems to solutions. These mappings highlight how solutions fail to comply with the selected problems. A problem (or an anomaly believed to be a problem) highlights, by definition, inconsistencies with the Current Solution Space. The formal representation (in terms of Kripke models) provides the basis to formally define the Problem Contextualisation.

Definition 5.9 (Problem Contextualisation) *Let $\mathcal{S}_t$ be the Current Solution Space (i.e., a collection of Kripke models). Let $\mathcal{P}_t$ be the Proposed System Problem Space (i.e., a collection of modal formulas). For each problem (i.e., modal formula) P^t_j in $\mathcal{P}_t$, by definition, exists a Kripke model $\mathcal{M}^t_i = \langle G^t_i, R^t_i, \Vdash^t_i \rangle$ in $\mathcal{S}_t$ such that $(\mathcal{M}^t_i, \Gamma) \nVdash^t_i P^t_j$ for some possible world Γ in G^t_i. That is, the problem P^t_j is invalid at the world Γ in the model $\mathcal{M}^t_i$.*

The mappings between the Current Solution Space $\mathcal{S}_t$ and the Proposed System Problem Space $\mathcal{P}_t$ (i.e., the relationship that comes from solutions looking for problems) identify requirements (demands, needs or desires of stakeholders) that correspond to problems as contextualised by (a part or all of) a current solution. These mappings represent the *objective requirements* or *functional requirements*.

Definition 5.10 (Objective Requirements) *Let $\mathcal{S}_t$ be the Current Solution Space and $\mathcal{P}_t$ be the Proposed System Problem Space. The objective requirements $\mathbf{R}_o^t$ consists of the mappings (i.e., pairs) that correspond to each problem P_j^t in $\mathcal{P}_t$ contextualised by a solution $\mathcal{M}_i^t$ in $\mathcal{S}_t$. Thus, for any possible world Γ in a Kripke model $\mathcal{M}_i^t \in \mathcal{S}_t$ and for any problem $P_j^t \in \mathcal{P}_t$ such that $(\mathcal{M}_i^t, \Gamma) \nVdash_i^t P_j^t$, the pair $\langle \Gamma, P_j^t \rangle$ belongs to $\mathbf{R}_o^t$. In formulae,*

$$\mathbf{R}_o^t = \left\{ \langle \Gamma, P_j^t \rangle | (\mathcal{M}_i^t, \Gamma) \nVdash_i^t P_j^t \right\} . \tag{5.2}$$

Example 5.3 (Clock continued) *Let consider P_j^t a problem in $\mathcal{P}_t$ be the formula:*

$$\Box R \rightarrow \Diamond R .$$

The formula is one of the basic axioms that define the standard modal logic that involves all Kripke frames without terminal worlds. That is, each possible world can reach another possible world by the accessibility relation [Barwise and Moss, 1996, Chagrov and Zakharyaschev, 1997]. Intuitively, the formula stresses that each world should always be able to access another world by the accessibility relation that identifies a repair transaction in the clock system. It is easy to see that P_j^t is true at the world Clock (Cl), but false at the world Broken Clock (BCl). That is,

$$\begin{aligned} (\mathcal{M}_i^t, Cl) &\Vdash_i^t \quad \Box R \rightarrow \Diamond R \\ (\mathcal{M}_i^t, BCl) &\nVdash_i^t \quad \Box R \rightarrow \Diamond R . \end{aligned}$$

At the world Clock $\Box R$ is false, because Cl can access itself (i.e., $Cl \Vdash T$) and BCl (i.e., $Cl \Vdash B$). Moreover, $Cl \nVdash R$ and $BCl \nVdash R$. Hence, it follows that $Cl \Vdash \Box R \rightarrow \Diamond R$. Similarly, it is possible to show that $BCl \nVdash \Box R \rightarrow \Diamond R$. At the world Broken Clock $\Box R$ is true and $\Diamond R$ is false, because BCl is unable to access any other world. Hence, $BCl \nVdash \Box R \rightarrow \Diamond R$. Figure 5.4 shows how the problem P_j^t is contextualised by the clock solution in $\mathcal{S}_t$. The possible world BCl contextualises the problem P_j^t. That is, $(\mathcal{M}_i^t, BCl) \nVdash_i^t P_j^t$. Hence, $\langle BCl, P_j^t \rangle$ belongs to $\mathbf{R}_o^t$.

5.2.4 Solution Space Transformation

The final step of the Solution Space Transformation consists of the reconciliation of the Solution Space $\mathcal{S}_t$ with the Proposed System Problem Space $\mathcal{P}_t$ into a Proposed

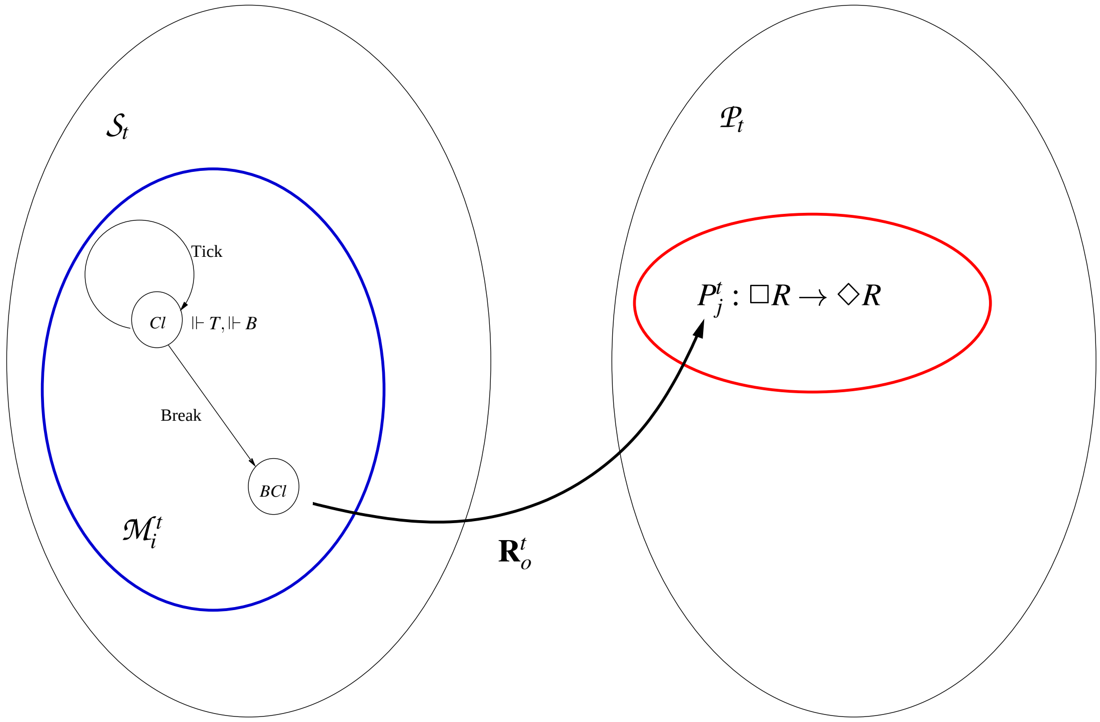


Figure 5.4: A problem contextualised by a solution.

Solution Space $\mathcal{S}_{t+1}$ (a subspace of a Future Solution Space $\mathcal{S}'$). The Proposed Solution Space $\mathcal{S}_{t+1}$ takes into account (or solve) the selected problems. The resolution of the selected problems identifies the proposed future solutions.

Definition 5.11 (Solution Space Transformation) *Let $\mathcal{S}_t$ be a Current Solution Space and $\mathcal{P}_t$ be a Proposed System Problem Space. Let $\mathcal{S}'$ and $\mathcal{S}_{t+1}$ be defined as follows.*

1. *A Future Solution Space, $\mathcal{S}'$, is an alternative local solution space postulated from the problem space analysis for the problem space $\mathcal{P}_t$. Normally, there are several alternative future solution spaces.*

2. *A Proposed Solution Space, $\mathcal{S}_{t+1}$, or simply a Proposed Solution, is a subspace of a Future Solution Space $\mathcal{S}'$ that includes the reconciliation of $\mathcal{S}_t$ with $\mathcal{P}_t$, i.e., $\mathcal{S}_t \rightarrow \mathcal{P}_t \rightarrow \mathcal{S}_{t+1}$.*

The Solution Space Transformation is the process of creating $\mathcal{S}_{t+1}$ from $\mathcal{S}_t$ by solving $\mathcal{P}_t$.

The reconciliation of $\mathcal{S}_t$ with $\mathcal{P}_t$ involves the resolution of the problems in $\mathcal{P}_t$. In logic terms, this means that the proposed solutions should satisfy the selected problems (or some of them). Note that the selected problems could be unsatisfiable as a whole (that is, any model is unable to satisfy all the formulas). This requires stakeholders to compromise (i.e., prioritise and refine) over the selected problems. The underlying logic framework allows us to identify model schemes that satisfy the selected problems. This requires to prove the validity of formulas by a proof system[7]. If a formula is satisfiable (that is, there exist a model in which the formula is valid), it would be possible to derive by the proof system a model (or counterexample) that satisfies the formula. The reconciliation finally forces the identified model schemes into future solutions.

Example 5.4 (Clock continued) *It is easy to show that the formula $\Box R \to \Diamond R$ holds on all Kripke frames without terminal worlds. That is, each possible world can always access another possible world. A Tableau system easily proves the validity of the formula $\Box R \to \Diamond R$. If the formula is satisfiable, there exist a closed tableau for the negated formula. The following tableau is a simple proof that the property $\Box R \to \Diamond R$ holds in all serial Kripke frames.*

1	$\neg(\Box R \to \Diamond R)$	(1)
1	$\Box R$	(2) *by conjunctive rule from (1)*
1	$\neg\Diamond R$	(3) *by conjunctive rule from (1)*
1	$\Diamond R$	(4) *by* **D** *special necessity rule from (2)*
1.1	R	(5) *By possibility rule from (4)*
1.1	$\neg R$	(6) *By basic necessity rule from (3)*

Hence, the future clock solutions should take into account the property represented by $\Box R \to \Diamond R$. Of course, there exist several solutions (devised from the initial clock) that fulfil this property. Figure 5.5, for instance, shows two clock (design) models without terminal worlds.

Both Kripke models reconcile the current solution space with the proposed system problem space. Thus, the two models would belong to the future solution space. In order to comply with the scheme $\Box R \to \Diamond R$, the two models are derived from the initial clock by adding further accessibility relations to the possible world Broken Clock, BCl. In this case it is easy to see that one of the possible solutions has an unfortunate

[7] Appendix B introduces a Tableau system for propositional modal logic.

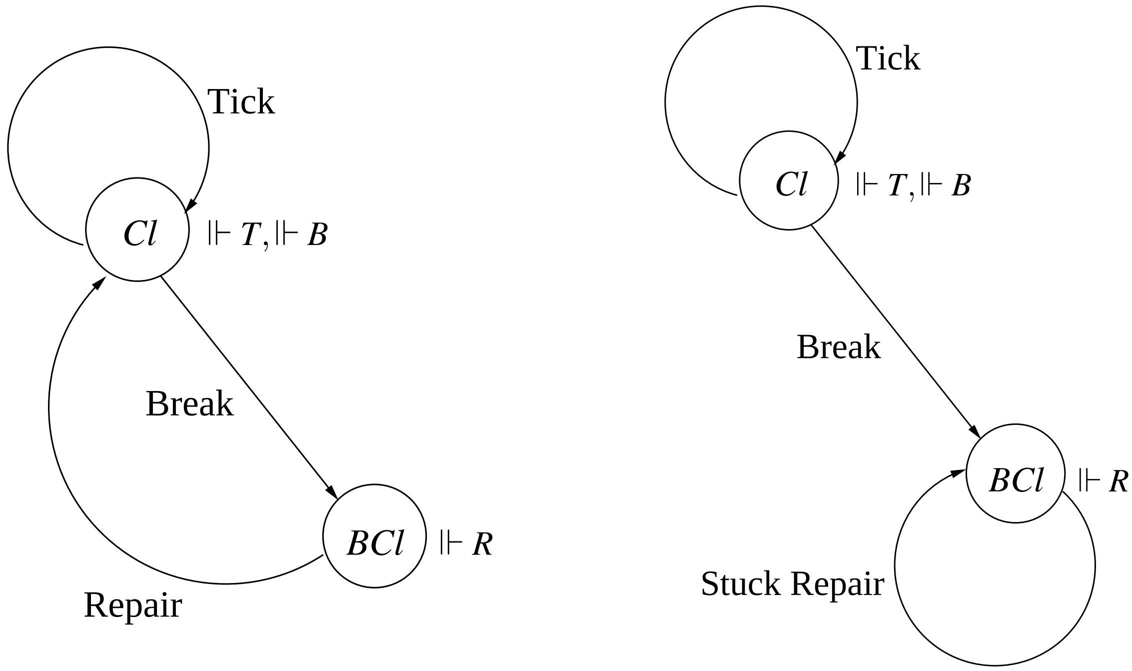


Figure 5.5: Two possible solutions that include the reconciliation of $\mathcal{S}_t$ with $\mathcal{P}_t$.

property. If the clock breaks down, it is beyond repair. That is, it is stuck in a broken status because BCl can only access itself. This simple case shows how design decisions may introduce flawed or undesired requirements. Any decision at the requirements stage affects future solutions.

Definition 5.12 (Problem Resolution) *Let $\mathcal{S}_t$ be the Current Solution Space, $\mathcal{P}_t$ be the Proposed System Problem Space and $\mathcal{S}_{t+1}$ be a Proposed Solution Space in a Future Solution Space $\mathcal{S}'$. Let consider $\langle \Gamma, P^t_j \rangle \in R^t_o$. Hence, $(\mathcal{M}^t_i, \Gamma) \not\Vdash^t_i P^t_j$ for some $\mathcal{M}^t_i$ in $\mathcal{S}_t$. A Kripke model $\mathcal{M}^{t+1}_i \in \mathcal{S}_{t+1}$, obtained by modifying the $\mathcal{M}^t_i$, solves the problem $P^t_j \in \mathcal{P}_t$ (that is, reconcile $\mathcal{S}_t$ with $\mathcal{P}_t$ by solving P^t_j into $\mathcal{S}_{t+1}$) if and only if $(\mathcal{M}^{t+1}_i, \Gamma) \Vdash^{t+1}_i P^t_j$.*

The final step of the Solution Space Transformation identifies mappings between the Proposed System Problem Space $\mathcal{P}_t$ and the Proposed Solution Space $\mathcal{S}_{t+1}$. These mappings of problems looking for solutions represent the *constraining requirements* or *non-functional requirements*.

Definition 5.13 (Constraining Requirements) *Let $\mathcal{S}_t$ be the Current Solution Space, $\mathcal{P}_t$ be the Proposed System Problem Space and $\mathcal{S}_{t+1}$ be a Proposed Solution Space in a Future Solution Space $\mathcal{S}'$. The constraining requirements $\mathbf{R}^t_c$ consists of the mappings* *(i.e., pairs) that correspond to each problem P^t_j in $\mathcal{P}_t$ solved by a solution $\mathcal{M}^{t+1}_i$ in $\mathcal{S}_{t+1}$. Thus, for any $\langle \Gamma, P^t_j \rangle \in \mathbf{R}^t_o$, and for any Kripke model $\mathcal{M}^{t+1}_i \in \mathcal{S}_{t+1}$ that solves the problem $P^t_j \in \mathcal{P}_t$, the pair $\langle P^t_j, \Gamma \rangle$ belongs to $\mathbf{R}^t_c$. In formulae,*

$$\mathbf{R}^t_c = \left\{ \langle P^t_j, \Gamma \rangle | (\Gamma, P^t_j) \in \mathbf{R}^t_o \textit{ and } (\mathcal{M}^{t+1}_i, \Gamma) \Vdash^{t+1}_i P^t_j \right\} . \tag{5.3}$$

Example 5.5 (Clock continued) *Figure 5.6 shows a solution space transformation for the clock. Notice the accurate selection of a repairable clock as future solution. The repairable clock $\mathcal{M}^{t+i}_i$ resolves the problem P^t_j by adding a repair transaction to the accessibility relation of $\mathcal{M}^t_i$.*

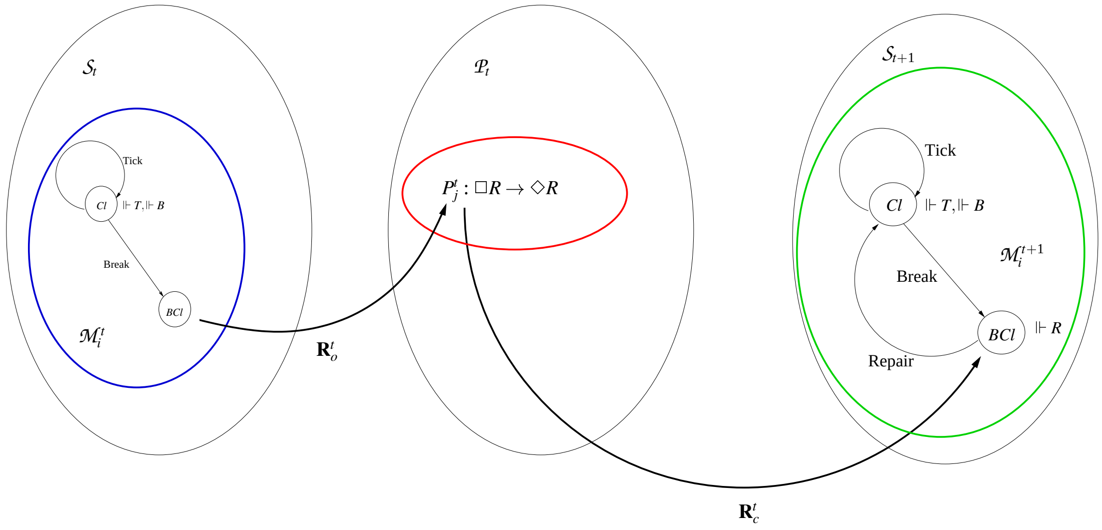


Figure 5.6: A solution space transformation.

5.2.5 Requirements Specification

The solution space transformation identifies the system *requirements specification* in terms of objective and constraining requirements. The system requirements specification consists of the collections of mappings between solutions and problems. The first part of a requirements specification consists of the objective requirements, which capture the relationship that comes from solutions looking for problems. The second part of a requirements specification consists of the constraining requirements, which capture how future solutions resolve given problems.

Definition 5.14 (Requirements Specification) *Requirements at any given time, t, can be represented as the set of all the arcs, that reflect the contextualised connections between the problem space and the current and future solution space. In formulae,*

$$\mathbf{RS}^t = (\mathbf{R}^t_o, \mathbf{R}^t_c) \tag{5.4}$$

This definition enables us further to interpret and understand requirements changes, hence requirements evolution.

5.3 Requirements Changes

The solution space transformation allows us the analysis of evolutionary aspects of requirements. Requirements, as mappings between solutions and problems, represent an account of the history of socio-technical issues arising and being solved during system production within industrial settings. The underlying heterogeneous account moreover provides a comprehensive viewpoint of system requirements. This holistic account allows the analysis of requirements changes with respect to solution and problem spaces. The analysis highlights and captures the mechanisms of requirements changes, hence requirements evolution. The formal extension of solution space transformation allows the modelling of requirements change, hence requirements change evolution.

There are various implications of the definition of solution space transformation. The solution space transformation represents requirements specifications in terms of mappings between solutions and problems. The mappings from solutions to contextualised problems identify objective (or functional) requirements. The mappings from problems to solutions whereas identify constraining (or non-functional) requirements. Thus, each solution space transformation identifies (a relationship network of) requirements. The mappings that represent requirements also identify requirements dependencies. Any change in objective requirements affect related constraining requirements. In general, this implies that diverse (types of) requirements affect each other. The heterogeneous account of solution space transformation highlights how diverse requirements, due to heterogeneous system parts (e.g., organisational structures, hardware and software components, procedures, etc.), may affect each other.

Example 5.6 (Allocation of Safety Requirements) *Safety requirements highlight dependencies between heterogeneous system parts (e.g., hardware and software). For instance, in the avionics case study presented in this thesis, the risk analysis of the hardware architecture points out safety system requirements. The system implementation constrains the allocation of these requirements. System integration usually provides further information in order to allocate some safety requirements. This results in pending request of changes to refine software requirements.*

Let imagine a scenario of hardware and software co-design. A current solution space consists of two solutions (or Kripke models) respectively for the hardware and software architectures. At some stage a proposed system problem space points out a system safety property. Both the hardware and software architecture contextualise (or fail to comply with) the safety property. There could be alternative future solutions that solve this problem. For instance, a future solution space takes into account the safety property by modifying the software architecture. This identifies mappings (i.e., constraining requirements) from the problem space to the software solution. Alternatively, another future solution space addresses the problem by modifying the hardware architecture. This identifies further hardware constraining requirements.

The requirements specification $\mathbf{RS}^t$ (i.e., the mappings $\mathbf{R}^t_o$ and $\mathbf{R}^t_c$) identifies many-to-many relationships between the contextualised problem space $\mathcal{P}_t$ and the current $\mathcal{S}_t$ and future $\mathcal{S}_{t+1}$ solution space. Sets of changes, as small as possible, in the problem and solution spaces could therefore cause non-linear, potentially explosive, change in the whole requirements specification $\mathbf{RS}^t$. This is the *cascade effect* of requirements changes. That is, any requirement, i.e., any mapping either in $\mathbf{R}^t_o$ or in $\mathbf{R}^t_c$, can affect or depend on other requirements. The impact of changes may therefore ripple through the requirements specification $\mathbf{RS}^t$ (i.e., the mappings $\mathbf{R}^t_o$ and $\mathbf{R}^t_c$) and affect different types of requirements. Stakeholders often fear the potentially devastating impact of changes. In order to avoid it, they get stuck in a *requirements paralysis*. That is, stakeholders avoid to change requirements that are likely to ripple cascade effects [Bergman et al., 2002a, Bergman et al., 2002b].

Example 5.7 (Cascade Effect) *The avionics case study presents occurrences of the cascade effect. The anomaly reports represent a rationale for requirements changes.*

Grouping requirements changes according to their rationale (i.e., relevant anomaly report) identifies requirements dependencies. These dependencies constrain the allocation of requirements changes to subsequent requirements releases. System implementation moreover provides further information in order to refine requirements. Stakeholders therefore prioritise requirements changes according to various environmental constraints (e.g., certification, testing, cost, etc.). Hence, change management allocates related requirements changes to subsequent releases. This results in the cascade effect. The inspection of the history of changes identifies these instances of the cascade effect.

Another implication of the solution space transformation is due to its requirements representation with respect to solutions, problems and stakeholders. Stakeholders judge whether solutions are available according to committed resources. Moreover, stakeholders select and prioritise the specific problems to be taken into account at a particular tile during system production. The combination of solutions and problems identifies requirements. Thus, on one hand stakeholders identify requirements. On the other hand, requirements identify stakeholders who own requirements. That is, any requirements shift highlights different viewpoints, hence stakeholders. It is therefore possible that stakeholders change while system production. Requirements definition involves different stakeholders at different project stages. For instance, the stakeholders (e.g., business stakeholders) involved at the beginning of a project are different than the ones (e.g., system users) involved at the end of it.

Example 5.8 (Processes, Stakeholders and Requirements) *The avionics and smart card case studies differently capture requirements. Although the two case studies are incomparable, because they are drawn from different industrial contexts, they provide different instances of requirements. Both case studies use a similar development process (i.e., the V model) tailored for the specific industrial context. The diverse product lines differently capture requirements. Each development environment identifies a unique trade-off among processes, stakeholders and system requirements. Any trade-off change ripples further changes among processes, stakeholders and system requirements. On one hand processes and stakeholders identify specific requirements, on the other hand any requirements shift corresponds to changes in processes or stake-*

holders. The better the understanding of relationships among processes, stakeholders and requirements, the better the ability to deal with requirements changes.

Finally, the solution space transformation allows the definition of requirements changes with respect to solutions and problems. The system requirements specification consists of collections of mappings between solutions and problems. Thus any solution or problem shift ripples requirements changes. Requirements changes therefore correspond to mapping changes. Hence, it is possible to capture requirements changes in terms of collection differences[8].

Definition 5.15 (Requirements Changes) *Let $\mathbf{RS}^t$ be the requirements specification corresponding to the current solution space transformation at time t. Let $\mathbf{RS}'$ be the requirements specification corresponding to a desired solution space transformation. At any time during system production requirements changes represent the gap between the current solution space transformation and a desired solution space transformation. In formulae,*

$$\begin{aligned} \mathbf{RC}^t &= \mathbf{RS}^t \ominus \mathbf{RS}' \\ &= \left(\mathbf{R}^t_o \ominus \mathbf{R}'_o, \mathbf{R}^t_c \ominus \mathbf{R}'_c\right) . \end{aligned} \tag{5.5}$$

5.4 Requirements Evolution

The solution space transformation captures requirements as mappings between solutions and problems. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings. This representation is useful to understand requirements changes. The solution space transformation describes the process of refining solutions in order to solve specific problems. Consecutive solution space transformations therefore describe the socio-technical evolution of solutions. Each sequence of solution space transformations captures how requirements have searched

[8]The set *symmetric difference* captures the differences between sets. The symmetric difference is the set of elements exclusively belonging to one set of two given sets. In formulae, $A \ominus B = (A - B) \cup (B - A)$. The *difference* of A and B is the set $A - B = \{a \,|\, a \in A \text{ and } a \notin B\}$. The *union* of A and B is the set $A \cup B = \{a \,|\, a \in A \text{ or } a \in B\}$.

solution spaces. On the other hand each sequence of solution space transformations identifies an instance of requirements evolution.

Definition 5.16 (Requirements Specification Evolution) *Let consider a sequence of solution space transformations. It captures the history of socio-technical problems arising and being solved. It moreover identifies an instance of the requirements specification evolution. In formulae,*

$$\mathbf{RS}\text{-EVOLUTION} = \left[\mathbf{RS}^1, \mathbf{RS}^2, \ldots, \mathbf{RS}^n\right]. \tag{5.6}$$

Example 5.9 (Clock continued) *Let assume that the self-loop in the clock solution is undesired. This is because, we would like to have a clock that does more than Tick. This implies that the property $\neg(T \rightarrow \Diamond T)$ belongs to the problem space. Future solutions should solve this problem by avoiding the Tick self-loop. The following closed tableau proves that the formula $(T \rightarrow \Diamond T)$ is valid in all reflexive frames. That is, the formula expresses a reflexive property on the accessibility of possible worlds in Kripke frames.*

1	$\neg(T \rightarrow \Diamond T)$	(1) *infers (2) and (3) by conjunctive rule*
1	T	(2)
1	$\neg\Diamond T$	(3)
1	$\neg T$	(4) *from (3) by **T** special necessity rule*

Hence, in order to enforce the property $\neg(T \rightarrow \Diamond T)$, future clock solutions would be without accessibility self-loops. Figure 5.7 shows another solution space transformation for the clock example.

Thus, the socio-technical evolution of the clock solutions consists of two consecutive solution space transformations. Figure 5.8 shows the entire sequence that represents the evolution of the clock solutions in order to solve the given problems.

Although it would have been possible to find an equivalent single solution space transformation for the same clock solution, solving problems requires stakeholders to commit resources within the development environment. Project risk is another reason that often justifies the resolution of problems over sequential solution space transformations. The solution space transformations identify the following clock requirements specification evolution

$$\text{CLOCK } \mathbf{RS}\text{-EVOLUTION} = \left[\mathbf{RS}^t, \mathbf{RS}^{t+1}\right].$$

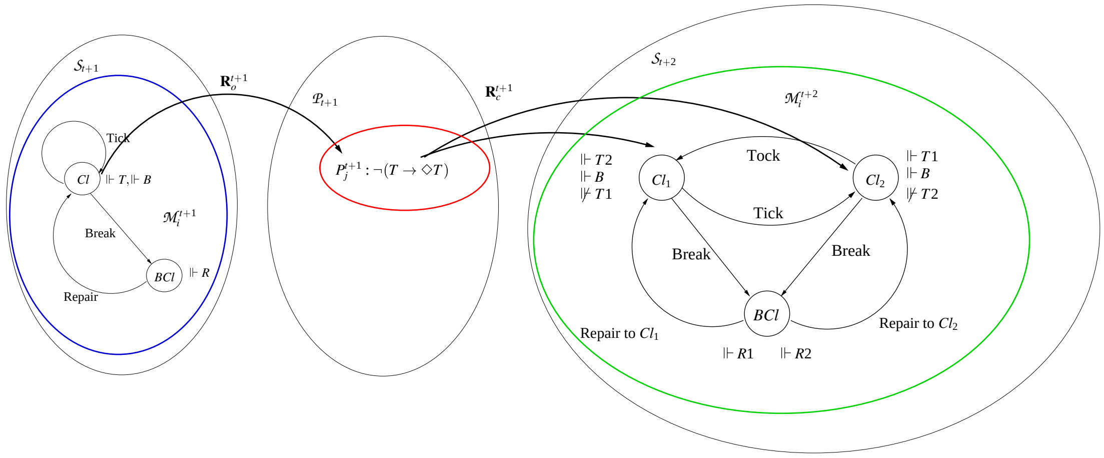


Figure 5.7: Another solution space transformation.

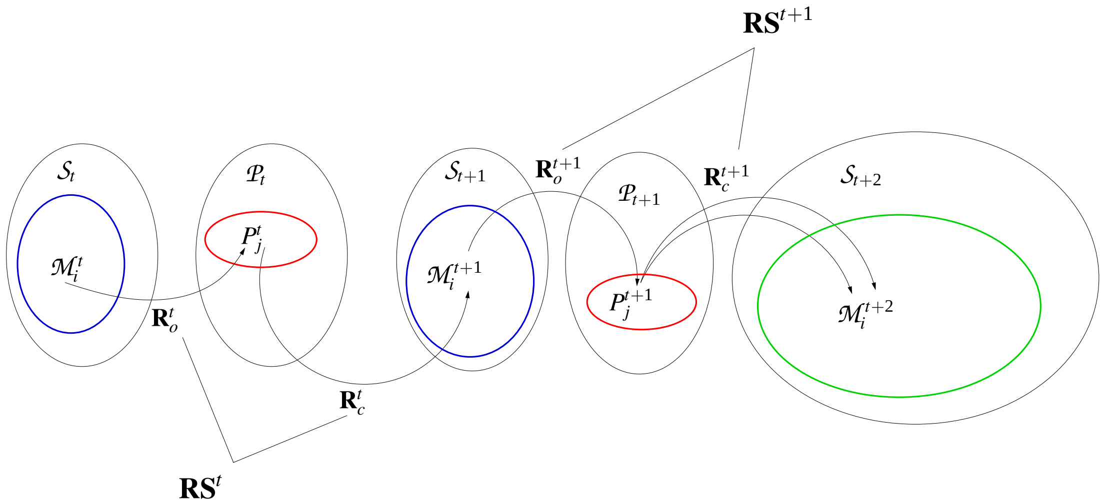


Figure 5.8: The entire sequence of solution space transformations.

The solution space transformation moreover allows the definition of requirements changes with respect to solutions and problems. Thus any solution or problem shift ripples requirements changes. Requirements changes evolution therefore captures those changes due to environmental evolution (e.g., changes in stakeholder knowledge or expectation).

Definition 5.17 (Requirements Changes Evolution) *Requirements Changes Evolution consists of the history of socio-technical evolution of bridging current solution* *space transformations to desired solution space transformations. In formulae,*

$$\mathbf{RC}\text{-EVOLUTION} = \left[\mathbf{RC}^1, \mathbf{RC}^2, \ldots, \mathbf{RC}^m\right]. \tag{5.7}$$

Definition 5.18 (Requirements Evolution) *Requirements Evolution is a co-evolutionary process. Requirements Evolution consists of the Requirements Specification Evolution and the Requirements Changes Evolution.*

5.5 Heterogeneous Requirements Evolution

Heterogeneous engineering considers system production as a whole. It provides a comprehensive account that stresses a holistic viewpoint, which allows us to understand the underlying mechanisms of evolution of socio-technical systems. Heterogeneous engineering is therefore convenient further to understand requirements processes. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings.

The formal extension of solution space transformation, a heterogeneous account of requirements, provides a framework to model and capture requirements evolution. The resulting framework is sufficient to interpret requirements changes. The formal framework captures how requirements evolve through consecutive solution space transformations. Hence, it is possible to define requirements evolution in terms of sequential solution space transformations. The characterisation of requirements and requirements changes allows the definition of requirements evolution. Requirements evolution consists of the requirements specification evolution and the requirements changes evolution. Hence, requirements evolution is a co-evolutionary process. *Heterogeneous Requirements Evolution* gives rise to new insights in requirements engineering.

A New Role for Requirements. Heterogeneous engineering stresses a different role for requirements. The shift from the paradigm of problems searching for solutions (i.e., *problem* $\rightarrow$ *solution*) to the one of solutions searching for problems (i.e., *solution* $\rightarrow$ *problem* $\rightarrow$ *solution*) points out a new role for requirements with respect to (design) solutions and problems. Figure 5.9 shows how the two different paradigms capture the relationships among requirements, design solutions and observed system problems.

a. Problems searching for solutions. b. Solutions searching for problems.

Figure 5.9: How the two different paradigms capture the relationships between requirements, design solutions and observed system problems.

Most software design processes and organisations rely on the first paradigm (i.e., problems searching for solutions). In this case, requirements represent problems to be solved by design solutions. Thus, software production takes into account a certain relationship between requirements, design and system implementation. This relationship implies a specific order to software production phases (e.g., first the collection of system requirements, then the design of solutions and finally the implementation, testing and so on). Regardless the adopted development process, each software production complies with the paradigm of problems searching for solutions. This is one of the reasons because most software development processes start with a requirement phase.

In contrast, heterogeneous engineering takes into account the second paradigm (i.e., solutions searching for problems). Heterogeneous engineering therefore points out that requirements link (design) solutions and given problems observed (by coding, testing, usage, etc.) in the system implementation. Requirements map solutions and problems. This implies a different role for requirements with respect to solutions and problems. On the one hand requirements map solutions to observed problems. On the other hand requirements narrow and browse solution spaces in order to address observed problems. The heterogeneous requirements role highlights new insights in the production of software systems.

Moreover, heterogeneous engineering helps us further to understand the mechanisms of requirements evolution. The modelling of requirements evolution highlights how requirements evolve due to the social shaping of socio-technical systems. On one hand the modelling supports the analysis of evolutionary phenomena (e.g., like in the avionics case study, stability, volatility, dependencies, etc.) in requirements, on the other hand the modelling supports the analysis of stakeholder interactions (e.g., like in the smart card case study, requirements viewpoints) in software production.

Implications for Requirements Processes and Tools. Heterogeneous engineering relies on a different paradigm. Heterogeneous engineering therefore highlights a new role for requirements (engineering) with respect to design solutions and observed system problems. This heterogeneous role has some implications for requirements processes as well as tools, in general, for software production.

Software production usually consists of the process of searching (or designing) solutions to given problems (or requirements). This implies that the requirements process has to search (or elicit) all system requirements in order to find the most suitable solution (by narrowing the solution space). System testing and verification therefore have to provide arguments that support system implementation, design and requirements. Therefore, in practice, verification and testing have to validate solutions by searching problems. In contrast, heterogeneous engineering highlights a new role for requirements. On the one hand the requirements process consists of matching solutions to observed problems. On the other hand the requirements process is to narrow and browse the solution space in order to address observed problems. Therefore, system testing and verification are to reveal problems that will be eventually matched to specific solutions by requirements.

The new role of requirements, with respect to solutions and problems, points out new scenarios of use for requirements engineering tools. Most requirements engineering tools support the maintenance of traceability between different software deliverables (e.g., requirements, change requests, rationale, design, etc.). However, future requirements engineering tools should also support the mapping of solutions to observed problems. That is, requirements engineering tools should support the analysis of observed problems in order to narrow the solution space. Thus, requirements engineering tools assume a major role in the analysis of observed problems in live production environments.

In summary, this chapter introduces a formal framework to model and capture requirements evolution. The framework relies on a heterogeneous account of requirements. Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings. The formal extension of solution space transformation defines a framework to model and capture requirements evolution. The resulting framework is sufficient to interpret requirements changes. The formal framework captures how requirements evolve through subsequent releases. Hence, it is possible to define requirements evolution in terms of sequential solution space transformations. The characterisation of requirements changes allows the definition of requirements evolution. Requirements evolution consists of the requirements specification evolution and the requirements changes evolution. Hence, requirements evolution is a co-evolutionary process.

Chapter 6

Capturing Evolutionary Requirements Dependencies

Requirements management methodologies and practices rely on requirements traceability. Although requirements traceability provides useful information about requirements, traceability manifests emergent evolutionary aspects just as requirements do. It is also important to understand requirements dependencies that constrain software production. Requirements dependencies, as an instance of traceability, identify relationships between requirements. Moreover, requirements dependencies constrain requirements evolution. Thus, it is important to capture these dependencies in order further to understand requirements evolution. This chapter shows how the formally augmented solution space transformation captures evolutionary requirements dependencies. Examples drawn from the avionics case study provide a realistic instance of requirements dependencies. These examples show how the heterogeneous framework captures evolutionary features of requirements, hence requirements evolution.

6.1 Requirements Traceability and Dependency

Requirements management methodologies and practices rely on requirements traceability [Sommerville and Sawyer, 1997a]. Although requirements traceability is crucial for requirements management, it is realistically difficult to decide which traceability information should be maintained. Traceability matrixes or tables maintain relevant requirements information. Requirements are entries matched with other elements in these representations (e.g., row or column entries). Traceability representations often assume that requirements are uniquely identified. Traceability practice requires that an organisation recognises the importance of requirements. Moreover, it has to establish well-defined policies to collect and maintain requirements. Unfortunately, traceability provides limited support for requirements management. There are various limitations that affect traceability practice.

6.1.1 Traceability Limitations

Practitioners often perceive that maintaining traceability increases workload. This is because the benefit of traceability is more evident in the long term rather than in the short term. However, maintaining traceability is expensive. Requirements changes moreover require to update traceability in order to record new or modified requirements dependencies. There are various limitations that affect traceability practice.

Scalability. Traceability representations (e.g., traceability matrixes or lists) are sensitive to the number of requirements. The more requirements, the less effective traceability. Although most requirements management tools keep traceability information, traceability effectively provides limited support as the number of requirements increase. One strategy to deal with numerous requirements is to group homogeneous requirements (e.g., functional requirements, subsystem requirements, etc.). This reduces the size of traceability representations, although it requires to refine traceability in hierarchical representations. Any grouping strategy therefore depends on suitable classifications of requirements. Unfortunately, requirements management tools provide limited support to identify requirements classifications tailored to specific industrial contexts.

Evolution. This affects requirements as well as traceability. Traceability representations need maintenance, otherwise they will become ineffective or, worst, useless. On one hand requirements changes affect traceability. On the other hand requirements changes provide further information about requirements dependencies. Although traceability takes into account requirements dependencies, these may evolve due to implementation (e.g., integration testing results) or environmental feedback (e.g., system usage). Traceability usually takes into account direct relationships between requirements. This provides limited support to capture indirect emerging dependencies[1]. Requirements changes may trigger subsequent changes into requirements. This results in a cascade effect of requirements changes. Thus, requirements dependencies emerge due to requirements changes. Traceability has therefore to reflect emerging dependencies.

Timeliness. Requirements traceability provides limited information about requirements timeliness. Although traceability identifies requirements dependencies, it fails to capture how these dependencies evolve during system production. Thus, traceability has a limited temporal validity. On the other hand it would be useful to know how requirements dependencies evolve throughout the system life cycle. Traceability supports the assessment of the impact of requirements changes. Empirical information about requirements would enhance traceability effectiveness. For instance, the likelihood of requirements changes may vary as the system development progresses. This type of information would be useful to refine any sensitivity analysis with respect to requirements changes. The refinement of traceability with timeliness information allows the gathering of emergent requirements dependencies.

6.1.2 Classification of Traceability

It is possible to classify traceability according to relationships with respect to requirements. There are four basic types of traceability: *Forward-to*, *Backward-from*, *Forward-from* and *Backward-to* [Jarke, 1998].

Forward-to requirements traceability links other documents, which may have pre-

[1]"Changes in goals will propagate downward through the levels while changes in the physical resources (such as faults or failures) will propagate upward. In other words, states can only be described as errors or faults with reference to their intended functional purpose. Thus reasons for proper function are derived *top-down*. In contrast, causes of improper function depend upon changes in the physical world (i.e., the implementation) and thus they are explained *bottom-up*.", [Leveson, 2000].

ceded the requirements document, to relevant requirements. Changes in stakeholder needs, as well as in technical assumptions, may require a radical reassessment of requirements relevance. *Backward-from requirements* traceability links requirements to their sources in other document or people. The contribution structures underlying requirements are crucial in validating requirements. These relationships identify *pre-traceability*, which consists of all the relationships between requirements and whatever (e.g., requirements rationale, requirements elicitation, requirements stakeholders, business contexts, etc.) precedes them. Table 6.1 shows examples of requirements pre-traceability [Sommerville and Sawyer, 1997a].

Table 6.1: Examples of requirements pre-traceability.

Traceability Type	**Description**
requirements-sources	Links the requirements and the people or documents which specified the requirements.
requirements-rationale	Links the requirements with a description of why that requirement has been specified.

Forward-from requirements traceability links requirements to design and implementation. This relationship allocates requirement responsibility to specific design and implementation components. It furthermore allows to assess the impact of requirements changes on design and implementation. *Backward-to requirements* traceability links design and implementation back to requirements. This relationship allows to assess whether system design and implementation comply with high-level requirements. It moreover allows to identify design or implementation for which requirements are under-specified or, worst, unspecified. These relationships identify *post-traceability*, which consists of all relationships between requirements and whatever (e.g., system design, implementation, testing results, system usage, etc.) follows them. Table 6.2 shows examples of requirements post-traceability [Sommerville and Sawyer, 1997a].

Requirements dependency represents a particular instance among the traceability types. It identifies relationships between requirements. The requirements-requirements traceability links the requirements with other requirements which are, in some way, de-

Table 6.2: Examples of requirements post-traceability.

Traceability Type	**Description**
requirements-architecture	Links requirements with the sub-systems that implement the requirements.
requirements-design	Links requirements with specific hardware or software components in the system which implement the requirements.
requirements-interface	Links requirements with the interfaces of external systems that provide the requirements.

pendent on them [Sommerville and Sawyer, 1997a]. Figure 6.1 shows a taxonomy of traceability.

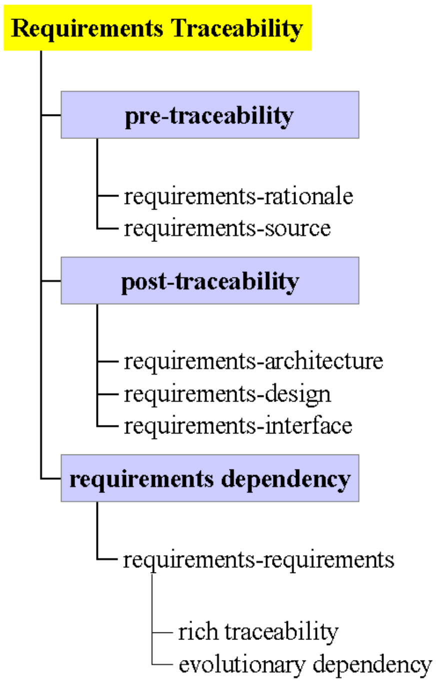


Figure 6.1: A taxonomy of requirements traceability.

6.1.3 Requirements Dependency

Requirements dependency is a peculiar type of traceability (see Figure 6.1). It identifies relationships between requirements. Understanding requirements dependency is very important in order to assess the impact of requirement changes. Among the requirements relationships are *Rich Traceability* and *Evolutionary Dependency*.

Rich Traceability [Hull et al., 2002] captures a *satisfaction argument* for each requirement. System requirements refine high-level user-requirements. Although low-level system requirements contribute towards the fulfilment of high-level user requirements, it is often difficult to assess the validity of these assertions. Thus, a satisfaction argument defines how overall low-level system requirements satisfy the high-level user requirements. There are two propositional operators: *conjunction* and *disjunction*. Conjunction indicates that the contribution of all refining system requirements is necessary for the user requirement satisfaction argument to hold. Disjunction indicates that the contribution of any one of the refining system requirements is necessary for the user requirement satisfaction argument to hold.

Notice that rich traceability gives rice to hierarchical refinements of requirements. This is similar to Intent Specifications [Leveson, 2000], which consist of multi-levels of requirement abstractions (from management level and system purpose level downwards to physical representation or code level and system operations level). The definition of hierarchies of requirements allows the reasoning at different level of abstractions[2]. Unfortunately, requirements changes affect high-level as well as low-level requirements in Intent Specifications. Moreover, requirements changes often propagate through different requirements levels[3]. Hence, it is very difficult to monitor and control the multi-level cascade effect of requirements changes. In accordance with

[2]"Hierarchy theory deals with the fundamental differences between one level of complexity and another. Its ultimate aim is to explain the relationships between different levels: what generates the levels, what separates them, and what links them. Emergent properties associated with a set of components at one level in an hierarchy are related to constraints upon the degree of freedom of those components.", [Leveson, 2000].

[3]"Mappings between levels are many-to-many: Components of the lower levels can serve several purposes while purposes at a higher level may be realised using several components of the lower-level model. These goal-oriented links between levels can be followed in either direction, reflecting either the means by which a function or goal can be accomplished (a link to the level below) or the goals or functions an object can affect (a link to the level above). So the means-ends hierarchy can be traversed in either a top-down (from ends to means) or bottom-up (from means to ends) direction.", [Leveson, 2000].

the notion of semantic coupling, Intent Specifications support strategies to reduce the cascade effect of changes [Weiss et al., 2003]. Although these strategies support the analysis and design of evolving systems, they provide limited support to understand the evolution of high-level system requirements. Thus the better our understanding of requirements evolution, the more effective design strategies. That is, understanding requirements evolution enhances our ability to inform and drive design strategies. Hence, evolution-informed strategies enhance our ability to design evolving systems.

Although traceability supports requirements management, it is unclear how requirements changes affect traceability. Requirements changes can affect traceability information to record new or modified dependencies. Hence, requirements dependencies (i.e., requirements-requirements traceability) may vary over time. In spite of this traceability fails to capture complex requirements dependencies due to changes. It is therefore useful to extend the notion of requirements dependency in order to capture emergent evolutionary behaviours, hence *Evolutionary Dependency*.

Definition 6.1 (Evolutionary Dependency) *Evolutionary Dependency identifies how changes eventually propagate through emergent requirements dependencies.*

Evolutionary dependency extends requirements-requirements traceability. It takes into account that requirements change over consecutive releases. Moreover, evolutionary dependency identifies how changes propagate through emergent, direct or indirect (e.g., testing results, implementation constraints, etc.), requirements dependencies. Evolutionary dependency therefore captures the fact that if changes affect some requirements, they will affect other requirements eventually. That is, how changes will manifest into requirements eventually. Evolutionary dependency therefore takes into account how requirements changes affect other requirements. Change rationale can trigger subsequent requirements changes. Requirements responses to change rationale refine evolutionary dependency. That is, the way changes spread over requirements represents a classification of evolutionary dependencies. It is possible to identify two general types: *single release* and *multiple release*. Single release changes affect a single requirements release. Whereas, multiple release changes affect subsequent requirement releases. This is because changes require further refinements or information. It is possible to further refine these two types as single or multiple requirements. It depends on whether requirements changes affect single or multiple (type of) requirements. This assumes that requirements group together homogeneously (e.g., functional requirements, subsystem requirements, component requirements, etc.). The most complex evolutionary dependency occurs as requirements changes affect multiple requirements over subsequent releases. In this case it is possible to have circular cascade effects. Requirements changes feedback (or refine) requirements through (circular) requirements dependencies. Figure 6.2 shows a taxonomy of evolutionary dependency.

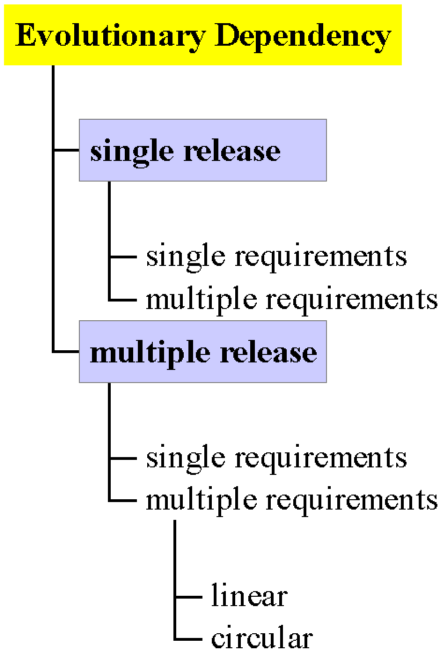


Figure 6.2: A taxonomy of evolutionary dependency.

6.2 Capturing Evolutionary Dependency

This section shows how the formal extension of solution space transformation captures instances of evolutionary dependencies drawn from the avionics case study. This provides another example of use of the proposed framework. The case study points out some basic dependencies. It is possible to represent these basic dependencies by simple Kripke models[4]. The solution space transformation then captures how dependencies emerge to create complex ones. This shows how formally augmented solution space transformations capture emergent requirements dependencies, hence evolutionary dependency.

6.2.1 Basic Dependencies

The empirical analysis of the avionics case study points out several instances of requirements dependencies. Looking at the rationale for changes allows the grouping of requirements changes. Moreover, it allows the identification of requirements dependencies. It is possible to refine complex dependencies in terms of basic ones. The case study highlights three basic dependencies: *Cascade Dependency*, *Self-loop Dependency* and *Refinement-loop Dependency*. These are instances of evolutionary dependencies.

Cascade Dependency. This is an instance of the cascade effect of requirements changes. It captures the fact that changes in some requirements trigger changes into other requirements eventually.

Example 6.1 (Cascade Dependency) *Let us consider the avionics case study. The requirements of the functions F1 and F2 manifest instances of cascade effects. Two anomaly reports required changes both in F1 and F2. The anomalies first triggered requirements changes in F1 and then in F2. Figure 6.3 shows a dependency graph for F1 and F2.*

The graph simply represents the evolutionary dependency between F1 and F2. The dependency relation, that is, the edge from F1 to F2, means that changes in F1 will trigger changes into F2 eventually. The relevant requirements changes were allocated in subsequent requirements releases. Hence, the two functions manifest a multiple-release multiple-requirement evolutionary dependency.

[4]Note that the Kripke models in the examples throughout this chapter present an overloading of names. The same names identify possible worlds as well as valid propositional letter at possible worlds. The names that identify nodes in the Kripke models identify possible worlds. Whereas, the names that follow the validity symbol $\Vdash$ are propositional letters valid at possible worlds.

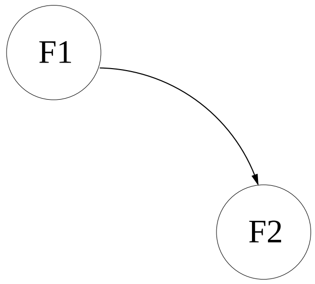


Figure 6.3: Evolutionary dependency graph for F1 and F2.

The analysis of the entire history of requirements allows the identification of complex dependencies between requirements. The self-loop and refinement-loop dependencies represent instances of complex cascade effects.

Self-loop Dependency. This identifies a self-dependence, that is, some requirements depend on themselves. This dependency implies that some changes require subsequent related refinements of requirements. Looking at the history of changes identifies related changes (i.e., due to the same rationale) that spread over subsequent releases and affect related requirements.

Example 6.2 (Self-loop Dependency) *The function F5 presents examples of self-loop dependencies. That is, a reported anomaly triggers requirements changes into subsequent releases of the requirements specification. Figure 6.4 shows a dependency graph that represents the self-loop dependency of F5. This represents an instance of multiple-release single-requirement evolutionary dependency.*

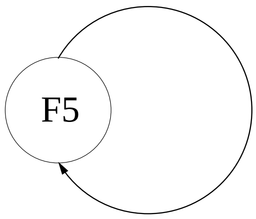


Figure 6.4: Evolutionary dependency graph for F5.

There are different reasons to allocate requirements changes over subsequent releases. For instance, stakeholders prioritise requirements changes according to environmental constraints (e.g., cost, certification, etc.). Another reason is that other development phases (e.g., implementation, testing, etc.) provide further information to refine some requirements.

Refinement-loop Dependency. This identifies mutual-dependencies over requirements. That is, changes in some requirements alternately trigger changes in other requirements and vice versa. This creates refinement loops of requirements changes. It looks like that stakeholders negotiate or mediate requirements through subsequent refinements. These dependency loops may emerge due to other development phases (e.g., system integration, system testing) that provide further information (e.g., implementation constraints) about requirements.

Example 6.3 (Refinement-loop Dependency) *The refinement-loop dependency is a combination of the cascade dependency with the self-loop dependency. Figure 6.5 shows a dependency graph that represents the refinement-loop dependency between F2 and F8. Changes, related by the same anomaly report, were alternately allocated to F2 and F8 over subsequent releases. This represents and instance of multiple-release multiple-requirements circular evolutionary dependency.*

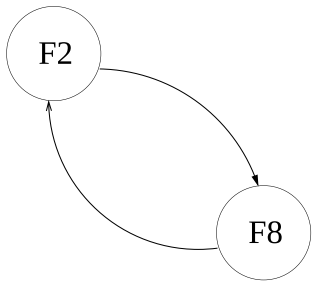


Figure 6.5: Evolutionary dependency graph for F2 and F8.

6.2.2 Modelling Dependencies

The avionics case study provides examples of requirements evolutionary dependencies. Evolutionary dependencies highlight how changes propagate into requirements. On one hand evolutionary dependencies highlight system features (e.g., dependencies due to the system architecture). On the other hand they point out that requirements evolve through consecutive releases, hence requirements evolution. Thus, evolutionary dependencies capture requirements evolution as well as system features. The formal extension of the solution space transformation allows the modelling of emergent evolutionary dependencies. Evolutionary dependencies populate solution spaces. Thus, solution spaces contain (Kripke) models of evolutionary dependencies. Whereas, requirement changes highlight emerging problems. A solution space transformation therefore resolves arising problems in future solutions. That is, it updates evolutionary dependencies in order to solve arising requirements changes and dependencies. The first step is to show how Kripke models easily capture the basic evolutionary dependencies.

Example 6.4 (Cascade Dependency continued) *The dependency graph for F1 and F2 simply is a Kripke structure. A function, which assigns propositional letters to possible worlds (i.e., nodes in the graph), extends a dependency graph to a Kripke model. This function defines a relationship between possible worlds and propositional letters. It mainly defines the validity of propositional letters at possible worlds. Figure 6.6 shows a Kripke model of the cascade dependency between F1 and F2.*

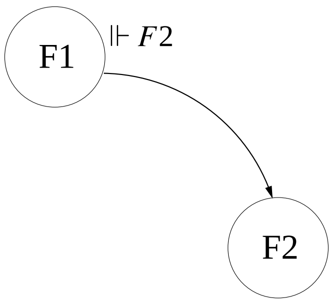


Figure 6.6: A Kripke model of the evolutionary dependency between F1 and F2.

The truth assignment corresponds to the accessibility relation (i.e., the edge of the graph). Thus, the propositional letter $F2$ is valid at the world (i.e., function) $F1$ in the proposed model, because $F2$ is accessible from $F1$. In other words, changes in F1 may trigger changes in F2 eventually. In this case F2 is a terminal possible world. That is, F2 in unable to access other possible worlds. This results in the fact that *every propositional letter is false at the possible world F2. In terms of evolutionary dependency, this means that changes in F2 are unable to affect other requirements.*

Notice that the evolutionary dependency graphs (models), this chapter shows, capture requirements dependencies at the functional level. That is, the dependency models represent how requirements changes propagate through system function requirements. This shows that the formal extension of the solution space transformation allows the modelling of change and evolution at different abstraction levels[5]. This complies with the features that other requirements engineering models highlight (e.g., [Hull et al., 2002, Leveson, 2000]). Requirements dependency models therefore capture how changes (due, for instance, to coding, testing, usage, etc.) in the physical dimension propagate upwards in the functional dimension [Leveson, 2000].

Example 6.5 (Self-loop and Refinement-loop Dependencies continued) *Similarly, it is possible to extend the dependency graphs for self-loop and refinement-loop dependencies to Kripke models. Figure 6.7 shows a Kripke model that represents the self-loop dependency of F5. Any reflexive Kripke model captures self-loop dependencies. On the other hand in any reflexive model each possible world has a reflexive loop [Barwise and Moss, 1996].*

Figure 6.8 shows a Kripke model for the refinement-loop dependency of F2 and F8. In this case F2 and F8 can access each other. This means that requirements changes alternately propagate into the two functions. Notice that, from a logic viewpoint, the two Kripke frames (see Figure 6.7 and 6.8) are bisimilar in the theory of non-well founded sets (or Hypersets) [Barwise and Moss, 1996].

The representation of basic evolutionary dependency is therefore straightforward. Simple conventions and notations easily capture requirements dependencies as Kripke models. It is possible to model complex dependencies as well. The combination (or composition) of the basic dependencies allow to capture complex ones. This results in the combination (or composition) of the underlying models[6].

[5]"In a means-ends abstraction, each level represents a different model of the same system. At any point in the hierarchy, the information at one level acts as the goals (the ends) with respect to the model at the next lower level (the means). Thus, in a means-ends abstraction, the current level specifies *what*, the level below *how*, and the level above *why*.", [Leveson, 2000].

[6]*Generation*, *Reduction* and *Disjoint Unison*, for instance, are three very important operations on modal logic models and frames which preserve truth and validity [Chagrov and Zakharyaschev, 1997].

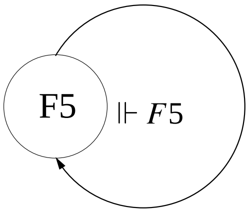


Figure 6.7: A Kripke model of the self-loop dependency for F5.

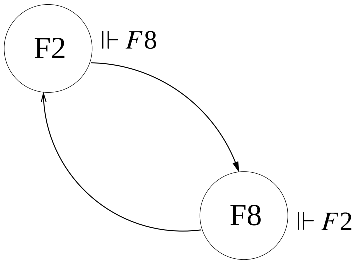


Figure 6.8: A Kripke model of the refinement-loop dependency between F2 and F8.

Example 6.6 (Complex Dependencies) *Figure 6.9 shows examples of complex dependencies identified in the avionics case study. Each complex dependency consists of a combination (or composition) of the three basic ones, i.e., cascade, self-loop and refinement-loop dependency. The truth values assignments will constrain the accessibility relationships of the Kripke frames.*

6.2.3 Capturing Emergent Dependencies

The formally augmented solution space transformation captures emergent evolutionary dependencies. That is, it is possible to capture how evolutionary dependencies change through solution space transformations. The idea is that solution spaces contain models of evolutionary dependencies. Whereas, anomalies as propositional modal formulas highlight dependency inconsistencies due to requirements changes. The solution space transformation therefore solves the arising problems (i.e., dependency inconsistencies) into proposed solution spaces. Hence, a sequence of solution space transformations captures emergent requirements dependencies. That is, it is possible to construct models of requirements dependencies using solution space transformations.

Example 6.7 (Cascade Dependency continued) *Let assume that the dependency between F1 and F2 is initially unknown. The initial Kripke model consists of two possible worlds, F1 and F2, without any accessibility relationship between them. This means*

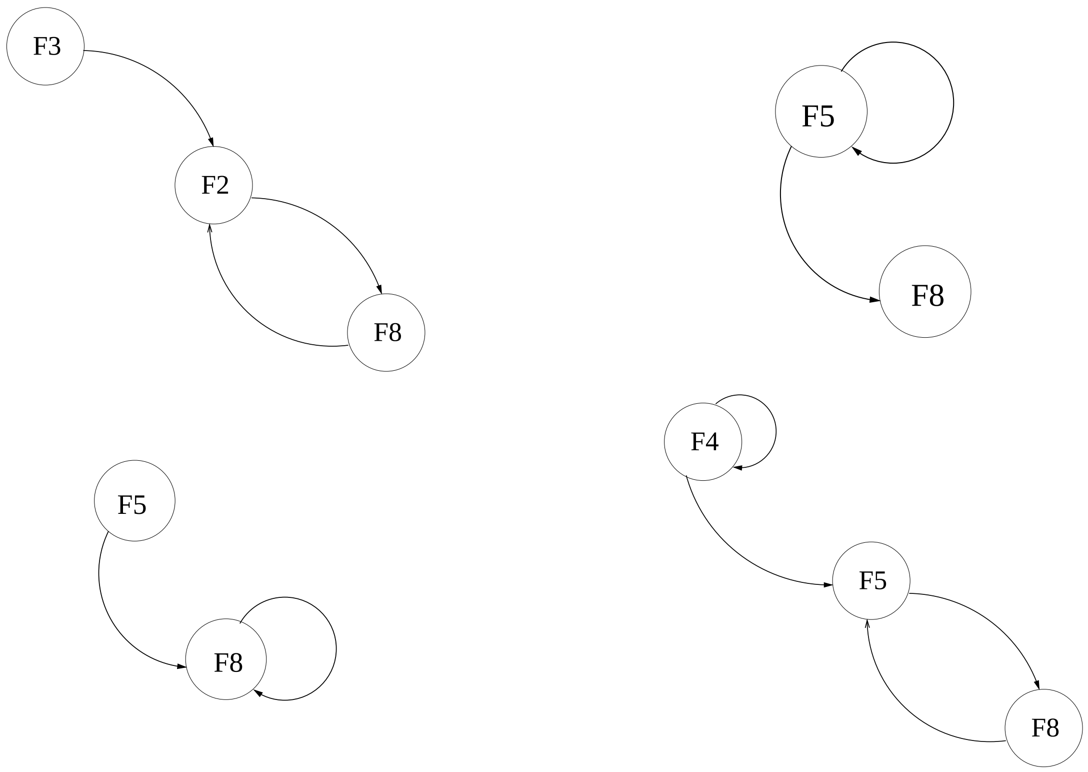


Figure 6.9: Examples of complex evolutionary dependencies.

that the possible worlds F1 and F2 are disconnected in the initial Kripke model. The dependency between F1 and F2 remains unchanged until an anomaly report triggers requirements changes that affect both of them. Stakeholders prioritise these requirements changes. They first allocate to a requirement release the changes for F1 and then to a future release the changes for F2. This results in a cascade dependency between F1 and F2. This situation highlights an anomaly (or inconsistency) with the current dependency model (i.e., a disconnected Kripke frame). In order to resolve this inconsistency, the proposed problem space contains the propositional modal formula

$$\Box F2 \rightarrow \Diamond F1 \, .$$

This formula means that "changes in F1 trigger changes into F2". It is easy to see that any disconnected Kripke frames fails to satisfy this formula, because $\Box F2$ *is true in any disconnected possible world and* $\Diamond F1$ *is false in any disconnected possible world. Notice that the given problem is similar to the axiom that characterises transitive Kripke frames (or simply frames without terminal worlds). A tableau can verify* *whether there exist a model that satisfies the given problem. This means to prove the validity of* $\neg(\Box F2 \rightarrow \Diamond F1)$. *The following tableau provides a countermodel for the given problem.*

1	$\neg(\Box F2 \rightarrow \Diamond F1)$	(1) *conjunctive rule giving (2) and (3)*
1	$\Box F2$	(2)
1	$\neg\Diamond F1$	(3)
1	$F2$	(4) *from (2) by* ***T*** *special necessity rule*
1	$\neg F1$	(5) *from (3) by* ***T*** *special necessity rule*

This tableau is open (i.e., it has an open branch). Thus, in this case, the only open branch (i.e., the entire tableau) provides a countermodel for the formula $\Box F2 \rightarrow \Diamond F1$. *A model for this formula can be any Kripke model that assigns the propositional truth values:* $\Vdash F2$ *and* $\nVdash F1$. *Notice that the validity of the propositional letter F2 indicates that there is an accessibility to the possible world F2. Figure 6.10 shows a solution space transformation that captures the resolution of the given problem. The possible world* $F1$ *complies with the formula* $\Box F2 \rightarrow \Diamond F1$. *Whereas, the possible world* $F2$ *fails to satisfy the same formula. This is because the proposed solution only takes into account the observed cascade effect that anomaly reports highlight.*

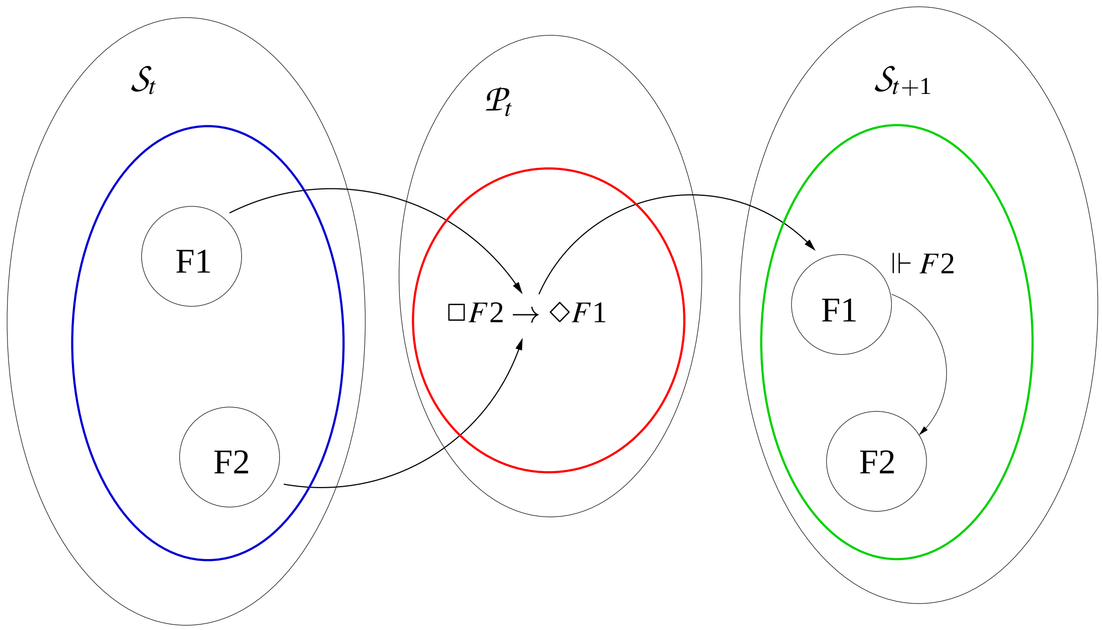


Figure 6.10: A solution space transformation for F1 and F2.

A future solution space transformation can also capture the evolutionary dependency between F2 and F8. Assume that there exist a refinement-loop dependency between F2 and F8. A solution space transformation will solve the new anomaly by extending the current solution space to a new proposed solution space. For instance, the propositional formulas $\Box F2 \rightarrow \Diamond F8$ and $\Box F8 \rightarrow \Diamond F2$ capture the given anomaly. Figure 6.11 shows a Kripke frame modelling the dependency between F1, F2 and F8. It represents a proposed solution.

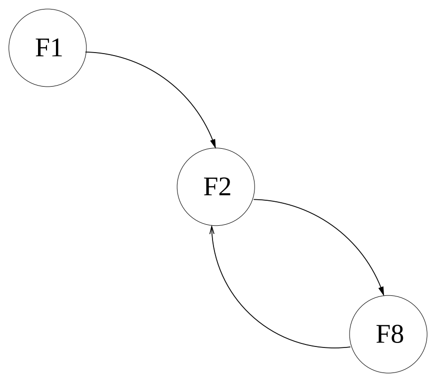


Figure 6.11: A Kripke frame that captures the dependency between F1, F2 and F8.

Similarly, consecutive solution space transformations can capture the evolutionary dependency for all the system functions. Figure 6.12 shows the observed dependencies.

6.2.4 Engineering Inferences

The evolutionary dependency models allow the gathering of engineering information. On the one hand the models capture the history of socio-technical issues arising and being solved within industrial settings. On the other hand it is possible to infer engineering information from the evolutionary dependency models. For instance, it is possible to enrich the semantics interpretation of the accessibility relation between functional requirements by associating weights with each pair of related possible worlds. Therefore, it would be possible to associate a cost for each relationship between two functions. Hence, it is possible to calculate the cost of propagating changes by summing the weights for all relationships between functions involved in particular requirements

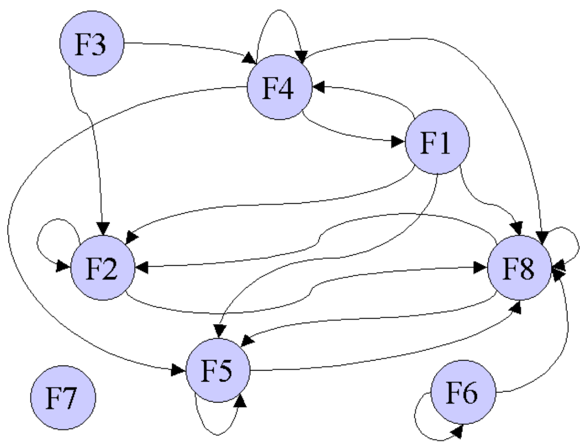


Figure 6.12: An example of evolutionary dependency graph.

change. Moreover, information about requirements evolution and volatility would allow the adjustment of cost models[7]. This information would enable the cost-effective management of requirements changes and the risk associated with them. However, the absence of a relationship from one function to another one could be interpreted as having a very expensive cost (e.g., infinite or non-affordable cost).

Example 6.8 *Figure 6.13 shows the evolutionary dependency model for F1, F2 and F8. It is possible to extend the models by labelling each transaction by the cost associated per each triggered requirements change. Thus, it is possible to calculate the cost of any change in F1 that triggers changes in F2 and F8 eventually. The cost of cascading changes is* w_1*,* w_2 *and* w_3 *for changes propagating from F1 to F2, from F2 to F8 and from F8 to F2, respectively. Therefore, if requirements exhibit the specific evolutionary dependency model (empirically constructed), the cost of implementing the associated changes would be* $n(w_1 + i(w2 + w3))$ *(where n is the number of changes in F1 that trigger changes in F2 and F8 eventually, and i is the number of times that changes are reiterated or negotiated between F2 and F8). Whereas, the accessibility from F2 to F1 (represented by a dashed arrow) would be very expensive, because it*

[7]"COCOMO II uses a factor called REVL, to adjust the effective size of the product caused by requirements evolution and volatility caused by such factors as mission or user interface evolution, technology upgrades, or COTS volatility.", [Boehm et al., 2000], p. 25.

will require changing the requirements of the software architecture (i.e., F1). Hence, although changes in F2 could affect F1, it is undesirable due to high cost and risk associated with changing F1.

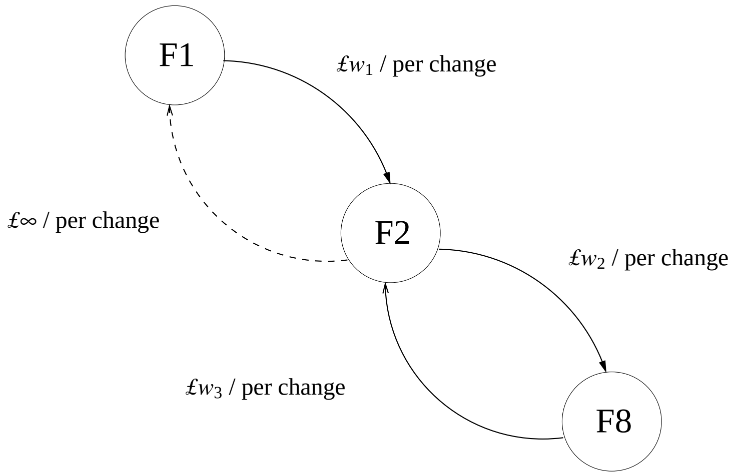


Figure 6.13: A weighted model of evolutionary dependencies.

6.3 Towards Requirements Evolution

This chapter shows how the formally augmented solution space transformation can capture emergent evolutionary dependency. The empirical analysis of the avionics case study highlights instances of evolutionary dependencies. The analysis points out three different basic dependencies: cascade, self-loop and refinement-loop dependency. The examples show that it is possible to capture evolutionary structures in terms of consecutive solution space transformations. Consecutive solution space transformations identify the history of problems arising and being solved. The underlying formal framework moreover allows the modelling of evolutionary dependency. This supports model-oriented software production.

The modelling of evolutionary dependency highlights that the formal extension of the solution space transformation enables the gathering of evolutionary information at different abstraction levels. Hence, the solution space transformation allows the modelling of different hierarchical features of requirements evolutions. This supports related requirements engineering approaches that rely on hierarchical refinements of requirements (e.g., Intent Specifications [Leveson, 2000]). The definition of hierarchies of requirements allows the reasoning at different level of abstractions. Unfortunately, requirements changes affect high-level as well as low-level requirements. Moreover, requirements changes often propagate through different requirements levels. Hence, the solution space transformation allows the reasoning of ripple effects of requirements changes at different abstraction levels. With respect to requirements hierarchies, the solution space transformation takes into account anomalies that relate to a lower level of abstraction. For instance, the solution space transformations, this chapter shows, allow the modelling of evolutionary requirements dependencies at the functional level. Although the problem spaces take into account requirements changes due to requirements refinements as well as anomalies at the physical level (e.g., coding and usage feedback).

In practice, the modelling of evolutionary requirements dependency and requirements evolution allows the reconciliation of solutions with observed anomalies. For instance, it would be possible to enhance the reasoning of evolutionary features of requirements, hence requirements evolution. Although most requirements engineering tools support the gathering of requirements (e.g., requirements management tools) and requirements changes (e.g., change management tools), they provide limited support in order to reasoning on observed evolutionary information. Hence, it is difficult to analyse and monitor emergent evolutionary features of requirements. Most requirements methodologies assess the impact of changes using traceability information. Unfortunately, changes affect traceability too. In contrast, the formal extension of solution space transformation allows the modelling of evolutionary requirements dependency, as mappings between dependency models and problems. This represents an account of the history of socio-technical issues arising and being solved within requirements hierarchies.

In summary, the formally augmented solution space transformation allows the gathering of emergent evolutionary features of solutions. Hence, heterogeneous requirements evolution provides the basic mechanisms to capture emergent evolutionary software production.

Chapter 7

Towards Requirements Evolution Engineering

Requirements represent an account of the history of socio-technical issues arising and being solved within industrial settings. Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of requirements evolution. Empirical analysis and requirements evolution modelling capture evolutionary aspects of system production. Accordingly, this chapter develops three main scenarios of practice: *Modelling Requirements Evolution*, *Process Calibration*, *Requirements Evolution Regression*. Moreover, it describes how heterogeneous requirements evolution supports the refinement of design models. Although these scenarios are descriptive, they provide an overall understanding how modelling requirements evolution enhances system production. The scenarios therefore further develop a rationale for *Requirements Evolution Engineering* (REE).

7.1 REE Rationale

Requirements evolution is a paradox in software production. On one hand requirements evolution is an emergent phenomenon that manifests during software production. On the other hand software production limitedly exploits requirements evolution. The empirical analyses of industrial case studies highlight several aspects of requirements evolution. For instance, it is possible to classify requirements according to their volatility and origins [PROTEUS, 1996]. Although empirical analyses successfully capture requirements evolution, they are context sensitive. That is, it is difficult to generalise results as well as methodologies. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings. Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. Hence, it is possible to model requirements evolution in terms of the formally augmented solution space transformation.

Empirical analysis and modelling therefore capture requirements evolution. They are convenient to identify requirements engineering practice. Although these methodologies provide a comprehensive account of requirements evolution, requirements engineering practice little exploits them. On the other hand requirements engineering practice needs to identify how requirements evolution supports software production. Requirements evolution engineering involves analysis, modelling and practice of requirements evolution. Requirements evolution therefore identifies strategies and methodologies that support software production. This thesis argues that the better the understanding and integration of requirements evolution, the more the support to system production. This chapter develops scenarios of use for requirements evolution engineering.

7.2 Observing Requirements Evolution

Rich data sources characterise software production environments. Unfortunately, it is very difficult to analyse rough data within live production environments. Engineering practice provides limited support for exploratory approaches of rich data sources. Although focusing on few environmental variables (e.g., maintenance cost, software reliability, etc.) aims to effectively establish a feedback to the software production, it inhibits the identification of emergent behaviours within software production.

Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings. Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. Hence, requirements evolution is an emergent aspect due to stakeholder interactions during software production. Thus, it is necessary to shape rough data in order to understand requirements evolution within live production environments.

The observation of evolutionary features within live production environments therefore requires to identify strategies to shape rough data in order to identify convenient viewpoints. These evolutionary viewpoints allow the gathering of requirements evolution. Thus, requirements evolution engineering involves those methodologies and strategies that allow the identification of convenient viewpoints for observing requirements evolution. Moreover, the overall goal is to establish and exploit requirements evolution feedback to software production.

Requirements are convenient to analyse how stakeholder interactions shape software systems throughout production. Empirical analyses highlight requirements evolution. Requirements evolution therefore is an emergent phenomenon that involves various aspects (e.g., development process, software product, etc.) of software production. It is therefore necessary to shape rough environmental data in order to capture evolutionary requirements features. Unfortunately, general requirements engineering practice provides limited guidance to tailor data analyses within live production environment. On the other hand data analyses provide valuable support to many requirements management activities. Among requirements engineering activities are requirements and change managements. The former, i.e., requirements management, consists of various tasks (e.g., changing requirements, editing requirements, updating traceability, etc.) that contribute to the overall management of requirements specifications. A well defined requirements management policy implements various guidelines [Sommerville and Sawyer, 1997a] that rely on specific requirements information (e.g., traceability). The latter, i.e., changes management, involves the tasks (e.g., maintaining changes rationale, assessing the impact of changes, etc.) that support the coherent organisation of the requests of changes due to arising anomalies. Most requirements engineering tools (e.g., Telelogic DOORS® and IBM Rational® RequisitePro®) support these two main activities, i.e., requirements and changes managements.

Requirements and changes managements constrain the interaction between requirements and changes. This interaction is very important to understand and shape requirements evolution. Requirements management policies identify strategies effectively to support the interaction between requirements and changes. For instance, the assessment of the impact of changes relies on requirements traceability. Thus, requirements management policies often require to maintain traceability in order to enhance sensitivity analyses (e.g., assessment of the impact of changes). Evolutionary requirements information further enhances the interaction between requirements and changes. Although management activities and policies identify strategies that organise the interaction between requirements and changes, they provide limited support to capture evolutionary information about requirements. Observing evolutionary features requires further resource commitments in order to capture relevant environmental information (e.g., taxonomy of changes, taxonomy of requirements, volatility, etc.). Figure 7.1 shows an example of evolutionary enhanced requirements information. It identifies (evolutionary) requirements information in terms of data sets and their interactions.

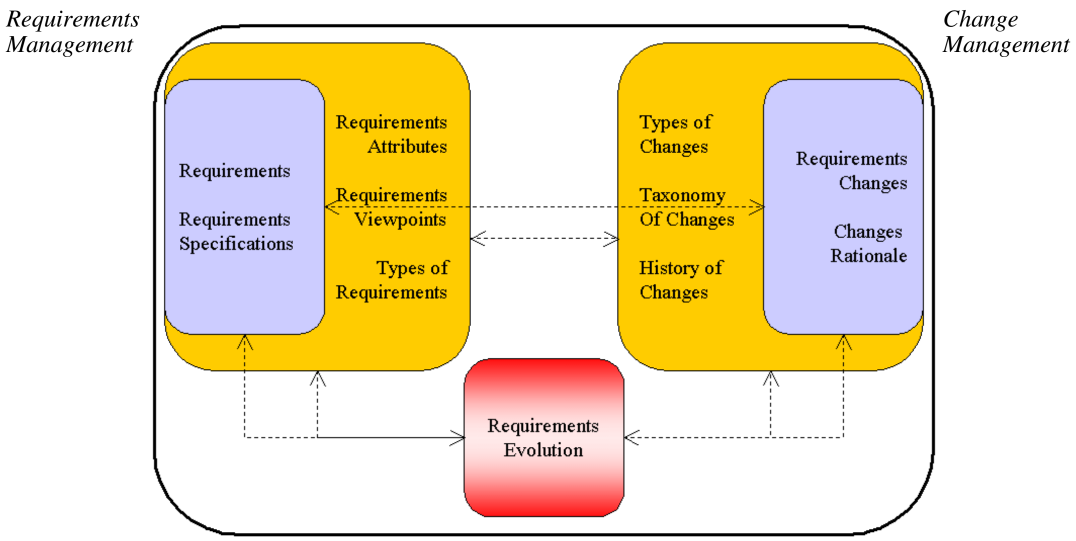


Figure 7.1: Evolutionary enhanced requirements information.

Table 7.1 describes the different evolutionary requirements information. For instance, a classification can capture whether stakeholders accepted or not requirements changes. A simple classification identifies three main types of changes: approved changes, rejected changes and pending requests of changes. Similarly, another classification describes the activities that change the requirements. Thus, maintenance activities fall into three categories: add, delete and modify requirements. Capturing this type of information supports the analysis of requirements and requirements changes, hence requirements evolution. For instance, the analysis of the change rationale (e.g., anomaly reports) can highlight emergent information about requirements evolution (e.g., evolutionary dependency).

Table 7.1: Evolutionary enhanced requirements information.

Data Set	**Description**
Requirements	Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings.
Requirements Specification	The specification consists of all active requirements, that is, mappings between (current and future) solutions and problems.
Requirements Attributes	These identify information that refines specific requirement features (e.g., functionality, traceability, dependency, etc.).
Requirements Viewpoints	Viewpoints identify specific perspectives (usually related to stakeholders) that are convenient to analyse requirements and their evolution.
Type of Requirement	Requirements classifications identify work practice as well as organisation features.
Requirements Change	Anomalies give rise to potential requirements changes. Stakeholders prioritise changes through a change management process that takes into account environmental constrains (e.g., cost).
Changes Rationale	Changes rationale supports requests of changes.
Type of Change	The classification of changes points out work practice in managing requirements changes as well as evolutionary system features.
Taxonomy of Change	It is possible to identify a taxonomy of changes with respect to the adopted classification of changes.
History of Change	The history of change consists of all changes approved for implementation in subsequent releases of the requirements specification.
Requirements Evolution	Requirements Evolution is a co-evolutionary process. Requirements Evolution consists of the Requirements Specification Evolution and the Requirements Changes Evolution.

Notice that there exist different relationships between evolutionary requirements information. For instance, any history of change affects several (one or more) releases of the requirements specification. The most general (i.e., many-to-many) relationships exist between changes rationale, requirements changes, requirements and requirements specification. For instance, a single anomaly can trigger subsequent requirements changes that affect multiple requirements (specifications). Therefore, capturing evolutionary requirements information highlights relationships that allow the analysis of requirements evolution. The collection of evolutionary requirements information enhances the systematic analysis of requirements evolution. Table 7.2 describes some examples of binary relationships.

Table 7.2: Examples of relationships between evolutionary requirements information.

Relationship	**Description**
Type - Size	The classification of measurable features according to types can provide further information to understand requirements evolution. For instance, the types of changes group requirements changes. This allows to identify whether there is a predominance of particular types of changes.
Size - Size	This relationship aims to identify any correlation between different measures of the same entity. For instance, it is possible to analyse the relationship between the number of requirements forming a functional specification and the number of requirements changes that occurred in the same functional specification. This points out whether there is any correlation between requirements and changes.
Type - Type	This relationship represent a combination of two Type-Size relationships. The relationship mainly combine two relationships related to different classifications. For instance, this may allow the identification of any relationship between requirements changes and software changes. This information would filter particular types of requirements changes in order to reduce corresponding types of software changes.

These relationships combine related evolutionary requirements information in order to identify any correlation between different information (or measures). Similarly, it is possible to identify other relationships that involve more than two evolutionary information. Although it is easy to combine evolutionary information, it is difficult to identify correlations that are valid across software production environments. On the other hand the systematic collection of evolutionary information supports the understanding of requirements evolution.

7.3 Scenarios of Use

Different development processes (e.g., V model, Spiral model, etc.) define how development activities (e.g., requirements elicitation, design, testing, etc.) interact each other and contribute towards software production. The monitoring of development activities provides feedback that supports process improvement [Bustard et al., 2000]. Different methodologies [Bustard et al., 2000] and standards (e.g., [Paulk et al., 1993]) rely on empirical analyses in order to address process improvement. On one hand processes are data sources. On the other hand empirical analyses provide quantitative evidence that captures system features. For instance, software reliability measures rely on the process ability to identify software faults. Similarly, requirements changes measures capture how changes management policies spread out changes over subsequent releases. Thus, capturing evolutionary requirements information provides useful feedback in development processes. This section articulates requirements evolution modelling in three scenarios of use: *Modelling Requirements Evolution*, *Process Calibration*, *Requirements Evolution Regression*.

Modelling Requirements Evolution. This scenario identifies a general process for modelling requirements evolution. Figure 7.2 shows the workflow of the scenario. Any development process leaves valuable information that represent environmental knowledge. This knowledge pervades production environments. Although production environments are rich data sources, most of the time data appears in many different artefacts (e.g., tools, reports, etc.) that make difficult to identify a comprehensive understanding of software production. However, empirical analyses allow the identification of evolutionary requirements features. For instance, it is possible to identify taxonomy of requirements and changes within production environments. Moreover, it is possible to classify requirements according to their likelihood of changing (e.g., volatile, stable, etc.). This information further supports requirements and change management activities. Notice that empirical analyses could also involve previous similar projects (e.g., product line projects) within production environments. Precious experience together with process information effectively allows the observation of evolutionary requirements information, hence requirements evolution. Finally, it is therefore possible to model requirements evolution. Requirements evolution, as defined, consists of requirements specification evolution and requirements changes evolution. These models support the monitoring of requirements evolution. Moreover, they capture evolutionary information (e.g., requirements evolutionary dependency) that supports the planning of iterative development phases.

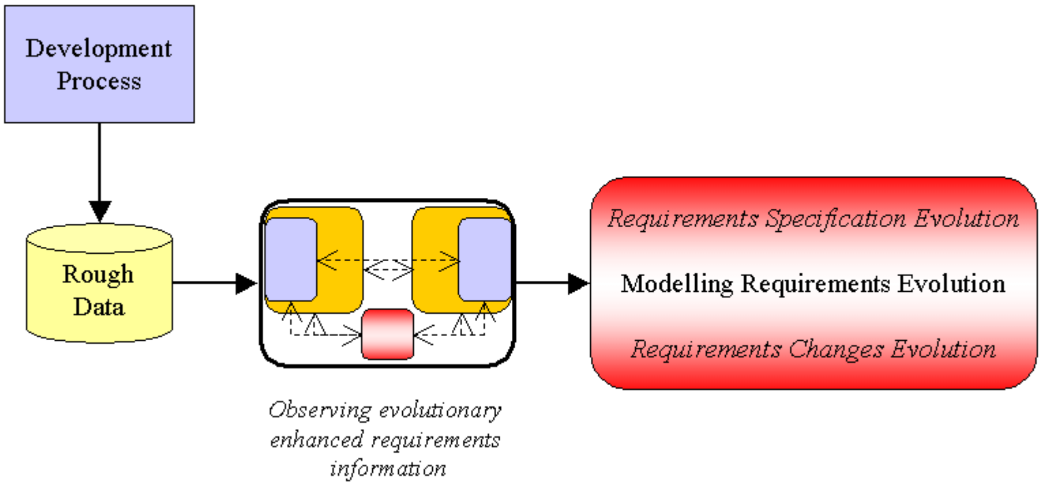


Figure 7.2: Modelling requirements evolution.

Process Calibration. Development processes organise software production in terms of development phases and activities. Although this allows the description of development processes, development phases and activities overlap each other. Moreover, as the software production progresses, emergent dependencies (e.g., requirements evolutionary dependencies) constrain the development process. This requires development processes to adjust accordingly. Figure 7.3 shows another scenario of use for requirements evolution modelling. The observations of evolutionary enhanced information and the requirements evolution models contribute towards the reconstruction of the actual development process. This can therefore provide feedback that is useful to calibrate (or adjust) development phases or activities.

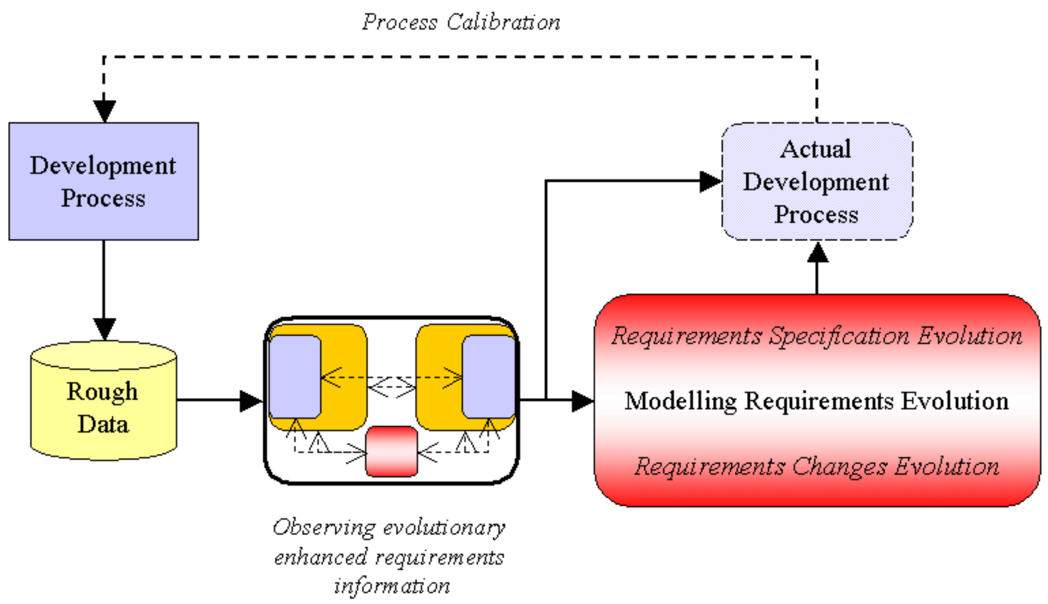


Figure 7.3: Process calibration.

This scenario supports process oriented methodologies too. Unfortunately, process oriented methodologies provide limited support to tailor the requirements process to specific software product (lines). This reduces the process effectiveness. Although process oriented methodologies plan development phases and activities, they provide limited control over the product. This limits the extent to which it is possible to adjust processes in order to enhance emergent product features. Moreover, this scenario is useful in product line contexts. For instance, evolutionary information and requirements evolution models highlight specific processes that mirror product line practice. Although product lines identify sets of similar projects, these projects differ each other on specific variability points [Bosch, 2000]. Thus, evolutionary requirements information and requirements evolution models highlight variability points that identify (new) product lines. The classification of requirements according to their stability (or volatility) allows the identification of the requirements for a new product line. For instance, the stable requirements identify the core of a new product line. Whereas the changeable requirements identify the variability points. This is very important in order to cost-effectively establish new product lines [Schmid and Verlage, 2002].

Requirements Evolution Regression. This scenario explores how evolutionary requirements information and requirements evolution models support the identification of emergent requirements features. Figure 7.4 shows the workflow of requirements evolution regression.

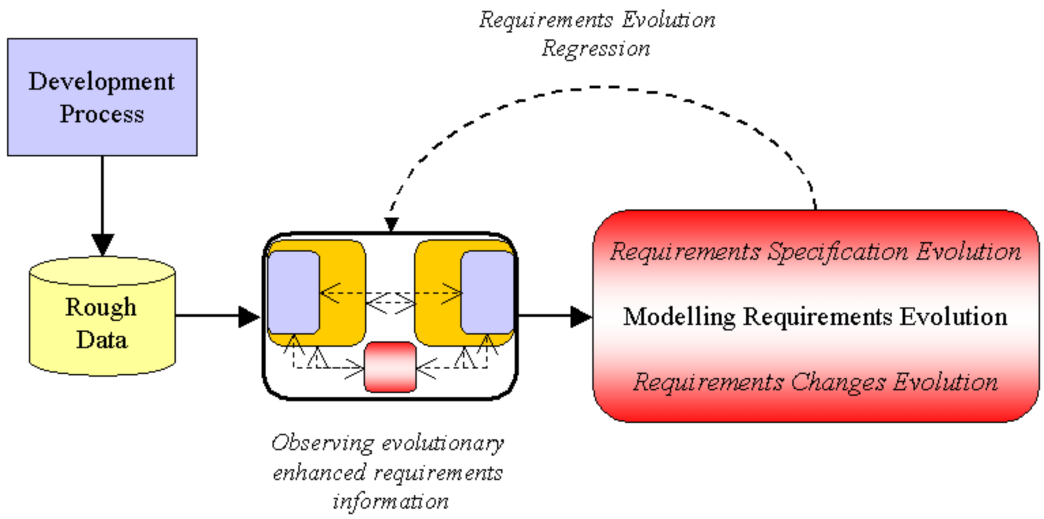


Figure 7.4: Requirements evolution regression.

Although general artefacts (e.g., classifications of requirements and changes) easily capture evolutionary requirements features, they evolve too. For instance, classifications capture work practice as well as product features. Thus, any change in work practice (e.g., a new change management policy) limits the applicability of classifications. This could result in misleading information. It is therefore necessary to monitor evolutionary requirements information throughout software production. Requirements evolution models support the monitoring of evolutionary requirements features. Observations that take into account the requirements evolution models define a requirements evolution regression, which supports the identification of emergent features (e.g., evolutionary requirements dependencies).

7.4 Evolutionary Design Observations

Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings. It is therefore possible to define requirements evolution in terms of sequential solution space transformations. The characterisation of requirements and requirements changes allows the definition of requirements evolution. Heterogeneous engineering stresses a different role for requirements. The shift from the paradigm of problems searching for solutions (i.e., $problem \rightarrow solution$) to the one of solutions searching for problems (i.e., $solution \rightarrow problem \rightarrow solution$) points out a new role for requirements with respect to (design) solutions and problems. Most software design processes and organisations rely on the first paradigm (i.e., problems searching for solutions). In this case, requirements represent problems to be solved by design solutions. Thus, software production takes into account a certain relationship between requirements, design and system implementation. This relationship implies a specific order to software production phases (e.g., first the collection of system requirements, then the solutions design and finally the implementation, testing and so on). Regardless the adopted development process, each software production complies with the paradigm of problems searching for solutions. This is one of the reasons because most software development processes start with a requirement phase. In contrast, heterogeneous engineering takes into account the second paradigm (i.e., solutions searching for problems). Heterogeneous engineering therefore points out that requirements link (design) solutions and problems observed (by coding, testing, usage, etc.) in the system under consideration. On the one hand requirements map solutions to given problems. This implies a different role for requirements with respect to solutions and problems. On the other hand requirements narrow and browse solution spaces in order to address observed problems. The heterogeneous requirements role highlights new insights in the production of software systems.

This section describes how the comprehensive account of heterogeneous requirements evolution supports the refinement of design models. Although design phases intend to identify the most suitable solutions that fulfil the given requirements. In contrast, heterogeneous requirements engineering highlights how design models (that is, solutions) support the observation of requirements (evolution). Moreover, the solution space transformation allows the gathering of requirements (evolution) during design. Consider a simple design scenario using UML [Rumbaugh et al., 1999]. UML is a popular modelling language that consists of different design levels (or viewpoints) and supports the implementation of (object-oriented) software systems. *Use cases* in UML models capture high level system requirements. Use cases identify system boundaries as well as system functionalities. Moreover, they identify the interactions between the system under development with other (external) systems or actors. In other words, "a use case describes sequences of actions a system performs that yield an observable result of value to a particular actor" [Leffingwell and Widrig, 2003]. Use cases in UML therefore represent the starting point with respect to the Rational Unified Process (RUP) [Hunt, 2000]. The RUP is an iterative framework that consists of different activities (e.g., requirements, design, etc.) articulated over different phases (i.e., inception, elaboration, construction and transition). The analysis of use cases provides the basis for the production of class diagrams [Bennett et al., 2001]. The class diagrams describe the basic building blocks of object-oriented systems. That is, the class diagrams show the classes that form a system and the collaborations (e.g., message passing) among those classes. Hence, class diagrams reflect an architectural view of the system under consideration. Figure 7.5, for instance, shows a design workflow as an activity diagram [Bennett et al., 2001]. The collaborations required between classes to provide the functional capability necessary to support requirements (in the form of use cases) will be examined in more details as the design progresses. The consideration of these collaborations will clarify the specification of the classes in the class model, hence class diagrams. As the specification becomes more and more firm, classes can be organised into subsystems of coherent and cohesive functional capability.

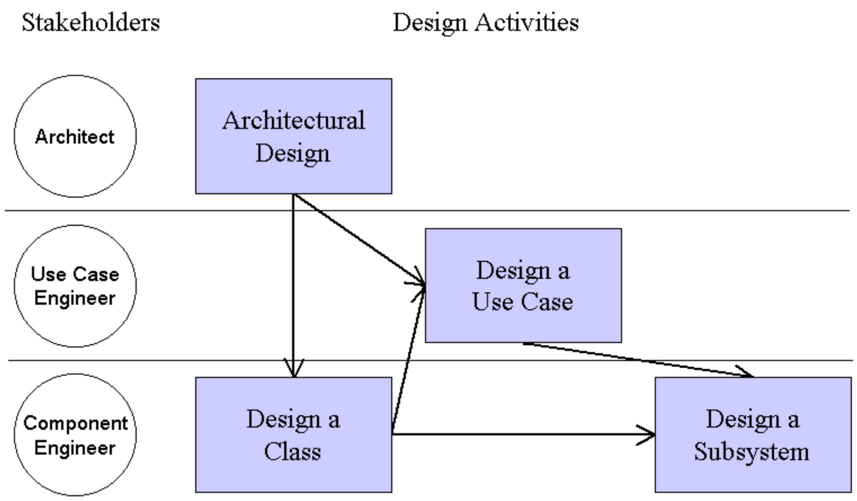


Figure 7.5: Design workflow as an activity diagram.

Thus, on one hand use cases capture system requirements, on the other hand system design identifies use cases. It is easy to figure out how the solution space transformation extends development workflows that rely on UML modelling. In this case, system design (in terms of architecture and classes) represents solutions. As development progresses, stakeholders highlight anomalies. According to the solution space transformation, requirements are mappings between solutions and problems. Figure 7.6 shows how the solution space transformation extends the design workflow. Solutions (i.e., architectural and classes design) contextualise given problems. The solution space transformation therefore resolves the given problems into the proposed future solutions. The future solutions reconcile the initial solutions with the given problems. The solution space transformation highlights how design solutions evolve in order to address observed problems. On the other hand the solution space transformation identifies the requirements, as mappings between solutions and problems. These requirements therefore highlight use cases of the system solutions. Hence, the solution space transformation supports the identification and refinement of use cases [Leffingwell and Widrig, 2003].

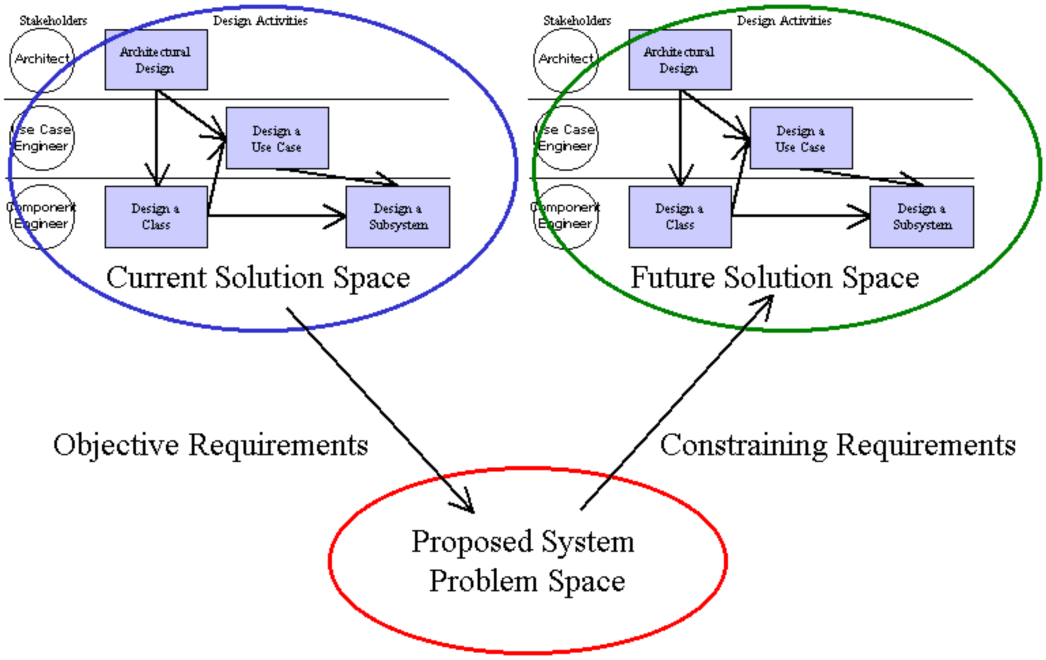


Figure 7.6: Extended design work flow using the solution space transformation.

7.5 Towards Requirements Evolution Engineering

The empirical analyses of industrial case studies highlight several aspects of requirements evolution. For instance, it is possible to classify requirements according to their volatility and origins. Although empirical analyses successfully capture requirements evolution, they are context sensitive. That is, it is difficult to generalise results as well as methodologies. Requirements evolution modelling allows to tailor development processes and artefacts to development environments. The combination of empirical analyses and requirements evolution models captures environmental features (e.g., work practice, product characterisation, etc.). This combination identifies a convenient requirements engineering practice that continuously provides feedback while software production progresses. Although these methodologies provide a comprehensive account of requirements evolution, requirements engineering practice little exploits them. On the other hand requirements engineering practice needs to identify how requirements evolution supports software production. Requirements evolution engineering involves analysis, modelling and practice of requirements evolution. Requirements evolution therefore identifies strategies and methodologies that support software production. This chapter presents three scenarios of practice: modelling requirements evolution, process calibration and requirements evolution regression. Moreover, it describes how the comprehensive account of heterogeneous requirements evolution supports the refinement of design models. Although design phases intend to identify the most suitable solutions that fulfil the given requirements. In contrast, heterogeneous requirements engineering highlights how design models (that is, solutions) support the observation of requirements evolution. The solution space transformation allows the gathering of requirements (evolution) during design. Although these scenarios are descriptive, they provide an overall understanding how modelling requirements evolution enhances system production. The scenarios therefore represent a contribution towards requirements evolution engineering.

Chapter 8

Taxonomy of Evolution and Dependability

Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings. However, requirements represent only one aspect of socio-technical evolution. Although evolution is a necessary feature of socio-technical systems, it often increases the risk of failures. This chapter reviews a taxonomy of evolution, as a conceptual framework for the analysis of socio-technical system evolution with respect to dependability. The identification of a broad spectrum of evolutions in socio-technical systems points out strong contingencies between system evolution and dependability. This thesis argues that the better our understanding of socio-technical evolution, the better system dependability. In summary, this chapter identifies a conceptual framework for the analysis of evolution and its influence on the dependability of socio-technical systems.

8.1 On Evolution and Dependability

Socio-technical systems [Coakes et al., 2000] are ubiquitous and pervasive in the modern electronic mediated society or information society. They support various activities in safety-critical contexts (e.g., Air Traffic Control, Medical Systems, Nuclear Power Plants, etc.). Although new socio-technical systems continuously arise, they mostly represent evolutions of existing systems. From an activity viewpoint, emerging socio-technical systems often support already existing activities. Thus, socio-technical systems mainly evolve (e.g., in terms of design, configuration, deployment, usage, etc.) in order to take into account environmental evolution. Software production captures to some extent socio-technical evolution by iterative development processes. On one hand evolution is inevitable and necessary for socio-technical systems. On the other hand evolution often affects system dependability. Unfortunately, a degradation in dependability, in the worst case, can cause catastrophic failures [Leveson, 1995, Perrow, 1999, Storey, 1996].

Heterogeneous engineering [MacKenzie, 1990] stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. These mechanisms highlight strong contingencies between system evolution and dependability. Unfortunately, the relationship between evolution and dependability has yet received limited attention. On the other hand both evolution and dependability are complex concepts. There are diverse definitions of evolution, although they regard specific aspects (e.g., software, architecture, etc.) of socio-technical systems. Moreover, they partially capture the evolution of socio-technical systems as a whole. Evolution can occur at different stages in the system life cycle, from early production stages (e.g., requirements evolution) to deployment, use and decommission (e.g., corrective or perfective maintenance). The existence of diverse (definitions of) evolutions is often misunderstanding. This gives rise to communication issues in production environments. Whereas, dependability is defined as *that property of a computer system such that reliance can justifiably be placed on the service it delivers. The service delivered by a system is its behaviour as perceived by its user(s). A user is another system (human or physical) interacting with the system considered* [Laprie, 1995, Laprie et al., 1998]. Different attributes[1] refine dependability according to complementary properties. The basic impairments of dependability define how *faults* (i.e., the initial cause) cause *er-*

[1]The dependability attributes are: Availability, Reliability, Safety, Confidentiality, Integrity and Maintainability. The association with Confidentiality of Integrity and Availability relative to the authorised actions leads to Security [Laprie et al., 1998].

rors (i.e., that parts of system states) that may lead to system *failures* (i.e., deviances of the system service). These identify the chain of mechanisms[2] (i.e., ..., fault, error, failure,...) by which system failures emerge. The *means*[3] for dependability are the methods or techniques that enhance the system ability to deliver the desired service and to place trust in this ability. Figure 8.1 shows the defined dependability tree [Laprie et al., 1998].

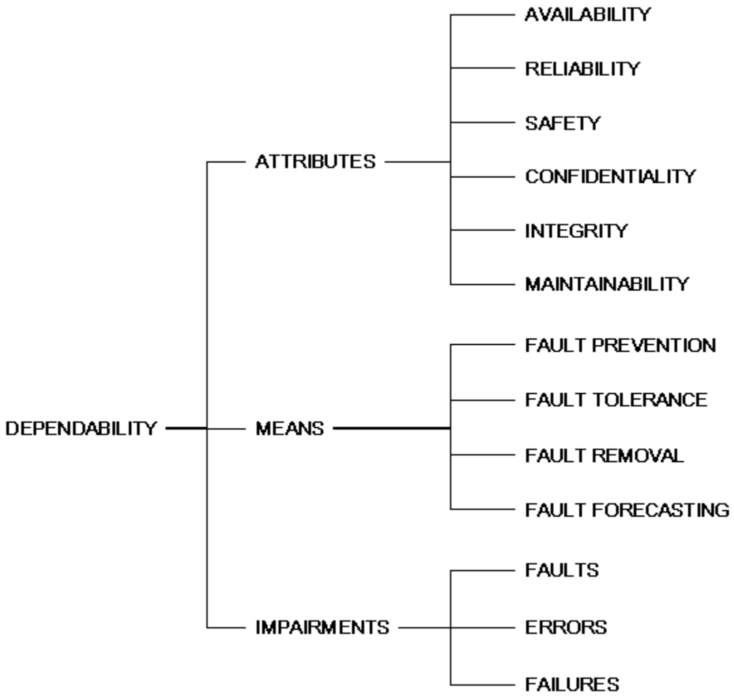


Figure 8.1: The dependability tree.

With respect to dependability the evolution of socio-technical systems transversely affect attributes, means and impairments. On one hand evolution can enhance dependability. On the other hand evolution can decrease system dependability. This chap-

[2]Note that it is possible to give slightly, but fundamental, different interpretations to these mechanisms. Different interpretations of the impairments and their mutual relationships highlight that failures emerge differently (e.g., ... error, fault, failure, ...) [Fenton and Pfleeger, 1996, Leveson, 1995].

[3]The means [Laprie et al., 1998] for dependability are: fault prevention, fault tolerance, fault removal and fault forecasting.

ter highlights emergent relationships between evolution and dependability of socio-technical systems. It reviews a taxonomy of evolution, as a conceptual framework for the analysis of socio-technical system evolution with respect to dependability. The identification of a broad spectrum of evolutions in socio-technical systems points out strong contingencies between system evolution and dependability. The taxonomy of evolution highlights how different evolutionary phenomena relate to dependability. This thesis argues that the better our understanding of socio-technical evolution, the better system dependability. In summary, this chapter identifies a conceptual framework for the analysis of evolution and its influence on the dependability of socio-technical systems.

8.2 Taxonomy of Evolution

Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings. However, requirements represent only one aspect of socio-technical evolution. This section reviews a taxonomy of evolution, as a conceptual framework for the analysis of the evolution of socio-technical systems. The evolutionary framework extends over two dimensions that define an evolutionary space for socio-technical systems. The two dimensions of the evolutionary space are: from *Evolution in Design* to *Evolution in Use* and from *Hard Evolution* to *Soft Evolution*.

Evolution in Design - Evolution in Use. This dimension captures the system life cycle perspective (or temporal dimension). System evolution can occur at different stages of the system life cycle. Evolution in design identifies technological evolution mainly due to designers and engineers and driven by technology innovations and financial constraints. With respect to technical systems, evolution in use identifies the social evolution due to social learning [Williams et al., 2000]. Social learning involves the process of fitting technological artefacts into existing socio-technical systems (i.e., heterogeneous networks of machines, systems, routines and culture) [Williams et al., 2000].

Hard Evolution - Soft Evolution. This dimension captures different system viewpoints in which evolution takes place (or physical dimension). Each viewpoint identifies different stakeholders. This dimension therefore reflects how stakeholders perceive different aspects of socio-technical systems. Hard[4] evolution identifies the evolution of technological artefacts (e.g., hardware and software). Whereas, soft[5] evolution identifies the social evolution (e.g., organisational evolution) with respect to these technological artefacts. Soft evolution therefore captures the evolution of stakeholder perception of technical systems.

These two dimensions (i.e., evolution in design - evolution in use and hard evolution - soft evolution) identify an evolutionary space for socio-technical systems. A point within this space identifies a trade-off between different socio-technical evolutions. The evolutionary space therefore captures the different evolutions that take place during the life cycle of socio-technical systems. Hence, the system life cycle describes a path within the evolutionary space [Williams et al., 2000]. The evolutionary space supports the analysis of evolution of socio-technical systems. The space easily identifies different evolutionary phenomena: *Software Evolution*, *Architecture (Design) Evolution*, *Requirements Evolution*, *Socio-technical System Evolution* and *Organisation Evolution*. These represent particular points within the evolutionary space. A software-centric view of socio-technical systems orders these points from software evolution to organisation evolution. Thus, software evolution is close to the origins of the space. Hence, the space identifies software evolution, as a combination of evolution

[4]"Hard systems viewpoints are basically those held by designers and engineers who are trying to create systems to meet an understood need in an effective and economic manner. Those in the soft camp caricature the approach as *head-down*, concerned with optimization, obsessed with quantitative metrics and highly pragmatic. So much so, in fact, that the term *system thinking* has been purloined by the soft camp as though they alone thought! The soft camp use the term *engineer's philosophy*, not too endearingly, to describe the hard approach, in which the requirement is stated by a customer and the engineer satisfies the requirement without question.", [Hitchins, 1992], p. 6.

[5]"Soft systems viewpoints are those held by behavioural, management, social anthropology, social psychology and other science students concerned with observing the living world, and in particular the human world. Human activity systems (HASs) are *messy*, in that they do not exhibit a clear need or purpose - if they can be said to exhibit purpose at all. Indeed, so complex is the real world of people that the idea of driving towards optimal solutions may be a non-starter - perhaps we should see if we can simply understand and concern ourselves with improving the situation.", [Hitchins, 1992], p. 7.

in design and hard evolution. Software evolution therefore takes into account evolution from a product viewpoint. Architecture (design) evolution describes how system design captures evolution. Requirements evolution represents an intermediate viewpoint. Requirements, as a means of stakeholder interaction, represent a central point that captures the evolution of socio-technical systems. Socio-technical system evolution takes into account evolution from a heterogeneous systemic viewpoint. Organisation evolution further emphasises the interaction between socio-technical systems and surrounding environments. Figure 8.2 shows the evolutionary space.

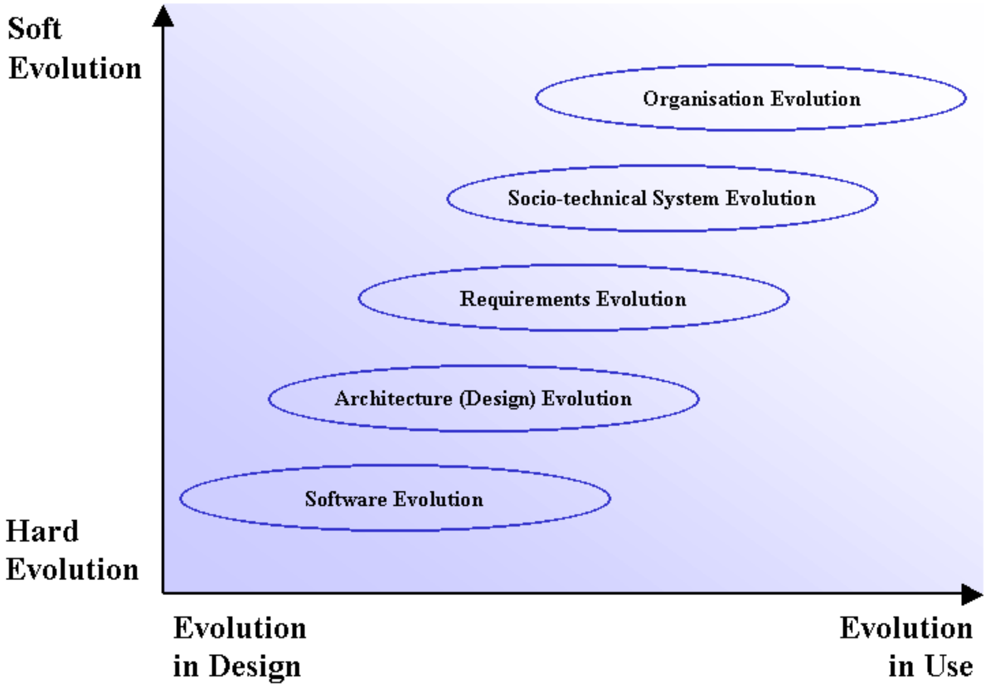


Figure 8.2: Evolutionary space for socio-technical systems.

Notice that these evolutionary phenomena define a simple classification of evolutions for socio-technical systems. These five different evolutionary phenomena have some similarities with other reference models (e.g., [Gunter et al., 2000, Perry, 1994]) that categorise and structure engineering aspects of socio-technical systems. The remainder of this section describes the different phenomena forming a taxonomy of evolution.

8.2.1 Software Evolution

Software evolution, as natural phenomenon of technical systems, identifies the *E-type programs* (i.e., software systems) [Lehman and Belady, 1985, Lehman et al., 1998]. According to the laws of software evolution [Lehman et al., 1998], E-type programs continuously evolve in order to be satisfactory (for users) and (need) to accommodate environmental changes. E-type programs have an increasing complexity if maintenance is neglected. They represent multi-level, multi-loop, multi-agent feedback systems. In spite of software engineering progress, managing evolution for such systems is still challenging [Lehman, 1998]. On the other hand management processes take into account software evolution (and in general evolution) as management issues [Weinberg, 1997] with a limited emphasis on evolutionary software features.

Recent research in object-oriented software [Foote and Yoder, 1996] highlights product features that explain some mechanisms of software evolution. It is possible to identify three main software evolution patterns: *Software Tectonics*, *Flexible Foundations* and *Metamorphosis*.

Software Tectonics emphasises that software systems need to accommodate arising changes. This involves fixing bugs as well as fixing flaws introduced early in the design. Hence software implementation should support evolution, otherwise software changes would just increase software complexity and destroy fundamental software structures. Moreover, in order to avoid software degradation, software systems should take into account requirements changes by series of small and controlled steps.

Flexible Foundations catalogues the need to construct systems out of stuff that can evolve along with them. That is, the basics (e.g., tools, languages, frameworks, etc.) of each system should be able to evolve themselves in order to support the software system evolution as a whole. Notice that the Flexible Foundations pattern focuses on the system's infrastructure (or architecture), regardless the specific implementation code.

Metamorphosis shows how equipping systems with mechanisms that allow them to dynamically manipulate their environments can help them better to integrate in

> these environments in order to fulfil evolving requirements. There exist different mechanisms that support the Metamorphosis pattern. For instance, extendible systems allow users to add new features. In contrast, mutable systems allow users to change their existing features.

In order to monitor software evolution, quantitative approaches like software metrics [Fenton and Pfleeger, 1996] capture evolutionary aspects that relate to quality software [IEEE, 1988a, IEEE, 1988b, ISO/IEC, 2001, Salamon and Wallace, 1994]. For instance, metrics quantify software changes in terms of Lines Of Code (LOC). Other metrics assess software reliability and take into account probabilistic models like *Reliability Growth Models* [Lyu, 1996]. These allow the prediction of software reliability trends. Reliability growth models differ each other on the basic assumptions (e.g., software maintenance is fault-free, operational profile, fault distribution, etc.). Unfortunately, these assumptions limitedly capture software production and affect the applicability of reliability growth models [Felici et al., 2000, Littlewood and Strigini, 2000].

Although quantitative approaches provide limited support to understand software evolution, recent research in empirical software engineering highlights that quantitative models capture specific evolutionary software features [Antoniol et al., 1999, Coleman et al., 1994, Graves et al., 2000, Kemerer and Slaughter, 1999]. For instance, in particular contexts like object-oriented systems, it would be possible to estimate the size of changes for evolving systems [Antoniol et al., 1999]. The quantitative model takes into account the classes changed (e.g., added or modified classes). Although software changes identify the affected parts of the software, they limitedly capture software evolution. It is therefore useful to look at the history of software changes as a whole. Sequence analysis allows to identify software processes in terms of changes [Kemerer and Slaughter, 1999]. Sequence analysis takes into account the history of changes and groups similar changes in phases. Therefore, it is possible to compare the evolution (in terms of sequences of changes) of different software projects. Unfortunately, the empirical studies of software evolution are still patchy due to the limited availability of rich evolutionary data repositories. The history of software changes also provides useful information for statistical models that evaluate the likelihood of faults introduced into modules [Graves et al., 2000]. These results further highlight the relationship between software changes and faults. Another experience with software metrics shows that simple quantitative models provide adequate support for decision making throughout the software life cycle. For instance, complexity metrics capture software structures and allow the evaluation of the maintainability of software systems [Coleman et al., 1994].

8.2.2 Architecture (Design) Evolution

With respect to evolution, at the design level, the role of the system architecture is unclear. Architecture evolution is risky, even if the evolutionary process and the architecture are well defined and understood [Kuusela, 1999, Sha et al., 1995]. On the other hand specific variability points capture to which extent product line architectures are able to adapt to future needs of system families [Bosch, 2000]. The ability to predict evolution is therefore crucial in the definition of product lines. The architecture therefore represents a trade-off between generality and specificity of product families.

Architecture evolution refines in different transformations according to transformation type and scope. Table 8.1 shows a taxonomy of software architecture transformations [Bosch, 2000]. The scope of architecture transformations extends to components as well as the architecture as a whole. The evolution of product line assets often involves a combination of architecture transformations [Bosch, 2000].

Table 8.1: A taxonomy of software architecture transformations.

SCOPE OF INPUT

TRANSFORMATION TYPE

	Component	**Architecture**
Added functionality, rules and/or constraints	Convert quality requirements to functionality	Impose architectural pattern
Restructuring	Apply design pattern	Impose architectural style

Although architecture evolution enhances system flexibility and ability to capture future arising requirements, it would be desirable to limit architecture evolution in particular cases. For instance, the empirical analysis of the avionics case study highlights that the architecture is a stable part of those systems that have stringent safety requirements [Anderson and Felici, 2000a]. This empirical result is in accordance with related research in requirements engineering that identifies the origins of stable requirements in the business core [PROTEUS, 1996], so is safety[6] for avionics contexts. The more constrained the business, the more evident the relationship between architecture stability and business core. On the other hand software stability highlights software architectures[7] in specific contexts (e.g, object-oriented) [Mens and Galan, 2002].

8.2.3 Requirements Evolution

Requirements evolution is a recognised phenomenon of socio-technical systems. Research and practice in requirements engineering takes it into account as a management problem, rather than a feature of socio-technical systems. On one hand empirical analyses highlight that requirements do evolve. Although it is unrealistic and impossible to freeze requirements, it is possible to analyse the extent to which requirements evolve. This allows the classification of requirements as stable or changing [PROTEUS, 1996]. On the other hand requirements engineering registers a multi-perspective shift towards model-driven software development [van Lamsweerde, 2000]. This motivates the search of structures that support the understanding and modelling of requirements evolution. In order to identify emergent evolutionary structures, it is necessary to be able to classify and identify evolutionary requirements and relationships. The combined process of classifying and relating entities is to some extent similar to the construction of other reference structures[8] (e.g., architectures, organisations, etc.) in socio-

[6]"The *safety culture* in an industry or organisation is the general attitude and approach to safety reflected by those who participate in that industry: management, workers, and government regulators. Major accidents often stem from flaws in this culture, expecially (1) overconfidence and complacency, (2) a disregard or low priority for safety, or (3) flawed resolution of conflicting goals.", [Leveson, 1995], p. 53.

[7]"...A software architecture is a collection of elements that share the same likelihood of change. Each category contains software elements that exhibit shared stability characteristics. ...A software architecture always contains a core layer that represents the hardest layer of change. It Identifies those features that cannot be changed without rebuilding the entire software system.", [Mens and Galan, 2002].

[8]"Architecture at its most elemental then is clustering and linking. For artefacts, clustering and linking are purposeful, where purposeful includes aesthetics. Entities, at this level of definition, may be

technical systems. It is possible to identify and relate requirements with respect to evolution, hence requirements evolution. For instance, types of requirements and requirements dependencies identify requirements information related to evolution.

Types of Requirements correspond to specific stakeholders involved in software production. Therefore, it is possible to classify requirements as stable and changing requirements. Each type of requirements has origin in specific entities within software organisations [PROTEUS, 1996]. The classification of requirements (as stable or changing) supports various strategies that deal with requirements changes. Among the possible strategies are *reducing changes*, *facilitating incorporation* of changes and *identifying needs* for changes as early as possible [PROTEUS, 1996]. These strategies use enabling technologies like predictive analysis of changes, traceability of requirements, formal framework for representing and reasoning about changes, and prototyping. Guidelines and checklists further support the strategies dealing with requirements changes.

Types of Changes capture work practice as well as requirements features with respect to evolution. For instance, a simple classification of changes (e.g., adding, deleting and modifying requirements) captures to some extent change management policy as well as product features. On one hand types of changes represent the implementation of a management policy defined within the production environment. On the other hand, the predominance of a particular type of change (e.g., adding new requirements) reflects how software production discovers system requirements. Moreover, quantitative approaches rely on the classification of changes in order to capture requirements evolution.

Requirements Traceability [Jarke, 1998] provides useful requirements information to assess the impact of changes. On one hand traceability supports the management of requirements. On the other hand it is necessary to maintain traceability too. Moreover, traceability represents a strategic policy that often involves an entire software organisation [Ramesh, 1998, Sommerville and Sawyer, 1997a]. Trace-

physical chunks, activities, people, ideas even. Architecture can be comprised of heterogeneous entities, so long as purposeful clustering can take place”, [Hitchins, 1992], p. 136.

ability maintenance involves various processes (e.g., change management policy) and tools (e.g., requirements management tools). These furthermore support the identification of relationships between software organisations and software products [Jarke, 1998]. Therefore, traceability identifies relationships between heterogeneous entities within software organisations.

These evolutionary information capture to some extent requirements evolution. The combination of evolutionary information together with modelling approaches (e.g., formal representation of requirements) supports the reasoning on requirements changes, hence requirements evolution. The further integration of quantitative approaches is still challenging for research and practice in requirements engineering. Unfortunately, the empirical analyses that monitor long term requirements evolution are still patchy.

8.2.4 Socio-technical System Evolution

Socio-technical systems involve heterogeneous entities (e.g., hardware, software, people, etc.) that relate each other. This type of system furthermore pervades and interacts within the surrounding environment. An holistic viewpoint is convenient to analyse the interactions between heterogeneous entities in socio-technical systems. These interactions capture some mechanisms of evolution for socio-technical systems. Figure 8.3, for instance, shows the SHEL model [Edwards, 1972].

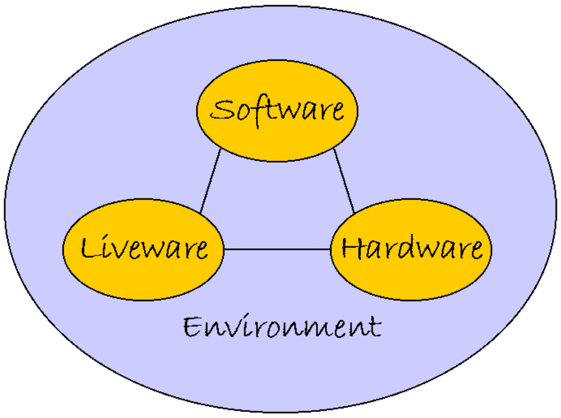


Figure 8.3: The SHEL model.

The SHEL model captures 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., system users, managers, etc.) resources embedded in a given *Environment* (e.g., socio-cultural, political, etc.). Thus, any process requires some knowledge that belongs to distributed heterogeneous system resources. Hence, a productive process consists of an instantiation of the SHEL model for a specific process execution. The holistic viewpoint of the SHEL model emphasises how drivers for software evolution often reside outside software artefacts. On the other hand evolution across resources allow new artefacts to emerge as resulting behaviour of the socio-technical system evolution.

Distributed Cognition [Norman, 1998] recognises the complex settings of socio-technical systems and 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 (e.g., rules, tools, representations, etc.) that 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 activities and artefacts are the two inseparable sides of the same phenomenon, that is, *human cognition*.

Although holistic models (like the SHEL model) are convenient to capture heterogeneous resources and structures that form socio-technical systems, descriptive models provide limited support to analyse socio-technical evolution. Understating the relationship between socio and technical evolution is still challenging for research and practice [Coakes et al., 2000]. Unfortunately, it is impossible to identify a single model that comprehensively captures the evolution of socio-technical systems. Various heterogeneous models together capture to some extent the evolution of socio-technical systems as a whole, although the combination of heterogeneous models often present practical issues. These issues are due to the gap that often exists between heterogeneous models. Although technical models support engineering activities (e.g., design, coding, testing, etc.), socio-technical systems are socially shaped

[MacKenzie and Wajcman, 1999]. The social shaping of technology involves the evolution of socio-technical systems. It is possible to explain the subtle mechanisms that explain the evolution of socio-technical systems, as adoption of technical artefacts. *Social Learning* [Williams et al., 2000], for instance, explains how human beings adopt technical systems in order to acquire computational artefacts, which allow them to accomplish specific tasks. Social Learning mainly involves two processes: *Innofusion* and *Domestication*.

Innofusion is a practical activity of learning by trying [Fleck, 1994]. In order to meet social needs, innofusion allows the customisation of computational artefacts provided by socio-technical systems. The underlying hypothesis is that system users, individually as well as collectively, develop more efficient ways of employing machineries through their usage experience. This kind of learning curve effect is well-known. Although it provides limited support for the initial introduction of new equipments, learning by doing improves the efficiency of system production over a long-term period of time.

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. It applies evolutionary metaphors of the generation of variations and selection of artefacts.

8.2.5 Organisation Evolution

Heterogeneous engineering [MacKenzie, 1990] provides a comprehensive account of system production. The system approach highlights that organisations mirror technical systems [Hughes and Hughes, 2000]. The socio-technical system account highlights the interaction between organisations and socio-technical systems. Figure 8.4 shows a simple socio-technical system model [O'Hara et al., 2000]. The left part of the model identifies the social system (or organisation). The social systems consists of (social) structures and people. The right part of the model identifies the technical system. The technical system consists of technology and tasks. The model capture to some extent the complex interaction between technology and people in an organisation.

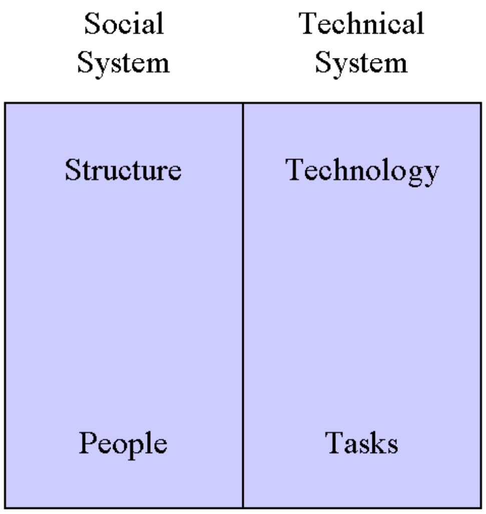


Figure 8.4: A simple socio-technical system model.

It is possible to use the simple socio-technical system model to analyse the impact of changes with respect to organisation. The model highlights how changes affect the technical system as well as the social system. Therefore, the model supports the classification of changes according to the affected resources (i.e., structures, people, technology or tasks). This points out how changes affect the interactions between resources (e.g., technology-tasks and technology-tasks-people). For instance, changes that involve only technology and tasks (e.g., technology upgrades) affect the interaction between these resources, but leave unmodified the interactions between the social system and the technical system. Although technology changes represent an hazard for organisations, they could be easy to plan. This depends on whether technology changes modify the way people accomplish their tasks. This is the case when changes in the technical system force new interactions between people. For example, people need to interact each other in order to accomplish their tasks using a new technology. The most complex changes are those that affect all the socio-technical entities (i.e., structures, people, technology and tasks). Although the socio-technical model captures the interactions due to changes, changes still represent an hazard for organisations. Unfortunately, the interactions between social systems and technical systems are very subtle. Failures to understand the dependencies within socio-technical systems are often the cause of organisational accidents [Reason, 1997].

8.3 On Dependability and Evolution

Dependability models capture evolution in different ways. For instance, fault tolerance models [Laprie et al., 1998, Randel, 2000] rely on failure distributions (e.g., Mean Time Between Failures) of systems. Monitoring this type of measure allows the characterisation of the evolution of system properties (e.g., reliability, availability, etc.). Probabilistic models [Lyu, 1996] may predict how dependability measures evolve according to the estimations of attributes and the assumptions about the operational profile of the system. In contrast, other models (e.g., [Littlewood et al., 2001, Littlewood and Strigini, 2000]) link dependability features with system structures and development processes. This allows the linking of failure profiles with design attributes (e.g., diversity) and system structures (e.g., redundancy). Structured models (e.g., FMEA, HAZOP, FTA) therefore assess the hazard related to system failures and their risk [Storey, 1996]. These models extend the *Domino* model, which assumes that an accident is the final result of a chain of events in a particular context [Heinrich, 1950]. Similarly, the *Cheese* model consists of different safety layers having evolving undependability holes. Hence, system failures completely arise and become catastrophically unrecoverable when they propagate through all the safety layers in place [Reason, 1997]. Despite these models capture diverse perspectives of the dynamics of system failures, they fail to capture evolution.

The evolutionary phenomena (e.g., software evolution, requirements evolution, etc.) of socio-technical systems differently contribute to dependability. The relationships between the evolutionary phenomena highlight a framework for the analysis of the evolution of socio-technical systems. Poor coordination between evolutionary phenomena may affect dependability. On the other hand evolutionary phenomena introduce diversity and may prevent system failures. Table 8.2 summarises the different dependability evolutionary perspectives and also proposes some engineering hints.

Table 8.2: Dependability perspectives of Evolution.

Evolution	**Dependability Perspective**	**Engineering Hint**
Software Evolution	Software evolution can affect dependability attributes. Nevertheless software evolution can improve dependability attributes by faults removal and maintenance to satisfy new arising requirements.	Monitor software complexity; Identify volatile software parts; Carefully manage basic software structures; Monitor dependability metrics.
Architecture (Design) Evolution	Architecture evolution is usually expensive and risky. If the evolution (process) is unclear, it could affect dependability. On the other hand the enhancement of system features (e.g., redundancy, performance, etc.) may require architecture evolution.	Assess the stability of software architecture; Understand the relationship between architecture and business core; Analyse any (proposed or implemented) architecture change.
Requirements Evolution	Requirements evolution could affect dependability. A non-effective management of changes may allow undesired changes that affect system dependability. On the other hand requirements evolution may enhance system dependability across subsequent releases.	Classify requirements according to their stability/volatility; Classify requirements changes; Monitor and model requirements evolution and dependencies.
Socio-technical System Evolution	System evolution may give rise to undependability. This is due to incomplete evolution of system resources. Hence, the interactions among resources serve to effectively deploy new system configurations. On the other hand human can react and learn how to deal with undependable situations. Unfortunately, continuous system changes may give rise to misunderstandings. Hence, human-computer interaction is an important aspect of system dependability.	Acquire a systemic view (i.e., Hardware, Software, Liveware and Environment); Monitor the interactions between resources; Understand evolutionary dependencies; Monitor and analyse the (human) activities supported by the system.
Organisation Evolution	Organisation evolution should reflect system evolution. Little coordination between system evolution and organisation evolution may give rise to undependability.	Understand environmental constraints; Understand the business culture; Identify obstacles to changes.

Example 8.1 *This example highlights how modelling requirements evolution allows the gathering of evolutionary aspects of socio-technical systems. For instance, the SHEL model [Edwards, 1972] points out that any system consists of diverse resources (i.e., Software, Hardware and Liveware). The interaction between these resources is critical for the functioning of systems. Moreover, changes occurring in some resources can affect the others. Therefore, it is very important to capture the dependencies between heterogeneous resources. Discrepancies between different resources may cause troublesome interactions, hence, trigger system failures. Modelling heterogeneous resources allow us to detect these discrepancies. For instance, it is possible to use model checking to discover mode confusions or automation surprises [Rushby, 2002]. These situations occur when computer systems behave differently than expected. It is possible to figure out how a solution space captures both system design models as well as mental models. Discrepancies between these models pinpoint design changes, or revision to training materials or procedures. On the other hand the solution space transformation captures how models need to change in order to solve arising problems or discrepancies. This scenario highlights how modelling requirements evolution captures evolutionary aspects of socio-technical systems. Moreover, it points out dependencies between heterogeneous parts of socio-technical systems. Therefore, modelling requirements evolution captures the evolution of socio-technical systems. These models can be further enriched by empirical data in order to identify the volatile or stable parts of socio-technical systems. The systematic modelling of requirements evolution combined with empirical analyses of evolutionary information would allow the understanding of the evolutionary nature of socio-technical systems. Enhancing our understanding of the evolution of socio-technical systems would provide valuable support to design.*

8.4 Evolution as Dependability

Software evolution represents just one aspect of the evolution of socio-technical systems. This chapter describes a taxonomy of evolution: *Software Evolution*, *Architecture (Design) Evolution*, *Requirements Evolution*, *Socio-technical System Evolution*, *Organisation Evolution*. The taxonomy identifies an evolutionary space, which provides a holistic viewpoint in order to analyse and understand the evolution of socio-technical systems. The taxonomy highlights the different aspects of the evolution of socio-technical systems. The taxonomy stresses the relationship between system evolution and dependability. Different models and methodologies take into account to some extent the evolution of socio-technical systems. Unfortunately, these models and methodologies rely on different assumptions about the evolution of socio-technical systems. This can cause misunderstandings and issues about system dependability and evolution, for instance:

Inconsistency. The basic assumptions of all adopted models (in order to characterise system dependability and evolution) may be inconsistent as a whole.

Coverage. The entire spectrum of models may be insufficient to cover all evolutionary aspects of system dependability.

Relational. The different evolutionary phenomena relate to each other. It is difficult to understand the different relationships between the evolutionary phenomena.

Emergent. New (or unexpected) system features may emerge from the different evolutionary phenomena.

In summary, the taxonomy of evolution represents a starting point for the analysis of socio-technical systems. It identifies a framework that allows the analysis of how socio-technical systems evolve. Moreover, the taxonomy provides a holistic viewpoint that identifies future directions for research and practice on system evolution with respect to system dependability. On one hand the resulting framework allows the classification of evolution of socio-technical systems. On the other hand the framework supports the analysis of the relationships between the different evolutionary phenomena with respect to dependability. Unfortunately, the collection and analysis of evolutionary data are very difficult activities, because evolutionary information is usually incomplete, distributed, unrelated and vaguely understood in complex industrial settings. The taxonomy of evolution points out that methodologies often rely on different assumptions of socio-technical system evolution. The dependability analysis with respect to evolution identifies a framework. The engineering hints related to each evolutionary phenomenon may serve as basics in order empirically to acquire a taxonomy of evolution. Future work aims to acquire practical experience using the taxonomy in industrial settings.

Chapter 9

Conclusions

Requirements Evolution is one of the main issues that affect development activities as well as system features (e.g., system dependability). Although researchers and practitioners recognise the importance of requirements evolution, research results and experience are still patchy. This points out a lack of methodologies that address requirements evolution. Requirements engineering research and practice mainly focus on management aspects. Management methodologies advocate process-oriented approaches in order to tackle requirements changes. On one hand these methodologies allow standardisation and organise work practice, although they provide limited support to tailor processes in order to capture system features. On the other hand system features pervade processes as well as organisations.

Requirements evolution is therefore an unavoidable feature of software production. Usually, requirements evolution is seen as an error in the engineering process. In contrast, this thesis takes into account requirements evolution as an essential feature of good design processes. Requirements evolution triggers a sequence of events, which allows changes to propagate throughout the development process. Regardless the iterative nature of software production, the definition of requirements is usually the first phase in popular development processes (e.g., V model and Spiral model). The requirements phase is therefore crucial for the success of software projects. This thesis considers software requirements evolution within industrial production environments. In contrast to the process-centred approach taken in current requirements engineering practice, this thesis takes a product-centred approach based on empirical analysis and modelling. 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. These case studies provide a detailed account of the cooperative processes adopted by stakeholders. The underlying hypothesis of this thesis is that stakeholder interaction in cooperative processes is a powerful driver of requirements evolution. This thesis addresses the lack of understanding about requirements evolution. It enhances our ability to understand requirements evolution. This thesis investigates the current understanding of requirements evolution and explores new directions in requirements evolution research. The empirical analysis of industrial case studies highlights software requirements evolution as an important issue. Unfortunately, traditional requirements engineering methodologies provide limited support to capture requirements evolution. Heterogeneous engineering provides a comprehensive account of system requirements. Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings. The formal extension of a heterogeneous account of requirements provides a framework to model and capture requirements evolution. The application of the proposed framework provides further evidence that it is possible to capture and model evolutionary information about requirements. The discussion of scenarios of use stresses practical necessities for methodologies addressing requirements evolution. Finally, the identification of a broad spectrum of evolutions in socio-technical systems points out strong contingencies between system evolution and dependability. This thesis argues that the better our understanding of socio-technical evolution, the better system dependability. In summary, this thesis is concerned with software requirements evolution in industrial settings. This thesis develops methodologies to empirically investigate and model requirements evolution, hence *Observational Models of Requirements Evolution*. The results provide new insights in requirements engineering and identify the foundations for requirements evolution engineering. This thesis addresses the problem of empirically understanding and modelling requirements evolution. The remainder of this chapter reviews the results in details.

9.1 Case Studies - Lessons Learned

9.1.1 Avionics Case Study

The case study enhances our understanding of requirements evolution drawn from industry. In summary, our experience is twofold. On the one hand we faced the practical problem of collecting evolutionary information. On the other hand we analysed the evolutionary features of requirements. These two main points are discussed in what follows.

9.1.1.1 Requirements Evolution Practice

The case study drawn from industry provides us some practical challenges. Similar challenges may arise in other industrial contexts. Behind any challenge there is actually an issue to deal with. The main practical challenges are: *Data Collection*, *Data Organisations and Goals*, and *Enhanced Visibility*.

Data Collection. Building a data repository of evolutionary data is a difficult task. There are various critical aspects that affect data collections under evolutionary scenarios. The collection of data should be well integrated into the development process. Poorly integrated data collection will result in increased workload and frustration for people who are supposed to collect data. Moreover, people will drop any data collection activity under the pressure of forthcoming deadlines. This will result in out of date data repositories. Substantial effort will be required in order to update these repositories during final stages of the development process. In the worst case the repositories will become unusable and ineffective. They will moreover fail to provide any evolutionary feedback in the development process.

Data Organisations and Goals. The organisation of evolutionary data is another aspect concerning an effective collection of evolutionary data. Data organisation affects our ability to analyse and identify evolutionary features. Unsuitable organisation will provide limited support to identify any emergent information. Data organisations should fulfil specific goals and address specific issues. Why are data collected? Is any data analysis foreseen? What are the expected outcomes? Who will review/read/use any (emergent) information? Answering these questions will help to organise an effective data repository. For instance, let assume that a simple history of changes is the main record of requirements changes. The history of requirements changes easily provides evidence of tracking changes for certification purpose, but it fails to provide any feedback in the development process. This is because it lacks any support to identify evolutionary relationships.

Enhanced Visibility. Issues relevant to requirements evolution affect project visibility. Poor coordination between different organisational layers may reduce the overall visibility within the project. Moreover, it affects our ability to assess the impact of changes (or to perform any sensitivity analysis). A trade-off between process and product management may tackle visibility issues. On the one hand process management is useful to standardise system developments, although it limits our visibility over software products. On the other hand product management enhances product features in development processes, although it affects process repeatability across different systems.

9.1.1.2 Requirements Evolution Features

The empirical analysis of the case study highlights evolutionary aspects of requirements. The requirements evolution features emphasise the specific case study, although it is possible to identify similar features in other case studies across different industrial contexts. The generality of the empirical investigation allows us easily to replicate the analyses. We summarise the main requirements evolution features in what follows.

Quantitative Requirements Evolution. The measurement of requirements evolution requires a well-defined standard policy to classifying requirements changes. Even a simple classification of requirements changes implies specific work practice and policy. For instance, the three-type classification of Added, Deleted and Modified requirements identifies the activities of: adding new requirements into the requirements specification, deleting requirements from the requirements specification and modifying requirements in the requirements specification. A management policy based on traceability information should support requirements management with respect to types of changes. For instance, requirements should be uniquely identified by an alphanumeric identification, which furthermore allows us uniquely to link changes and requirements. The combination of type of change together with traceability information allows us to measure various aspects of requirements evolution. Moreover, the combination of traceability and type of change easily supports the analysis of the impact of changes. Although the impact of changes is critical for project management, it needs subsequent refinements that take into account various requirements aspects (e.g., changes criticality, type of change, history of changes, etc.).

Taxonomy of Requirements Changes. The investigation of the history of changes points out a taxonomy of requirements changes. The resulting taxonomy characterises the specific case study as well as its industrial context. The identification of a taxonomy of requirements changes supports the introduction of standard work practice in the development environment. On the one hand a taxonomy will help to reduce biases due to different experiences and expertise. Moreover, a taxonomy will support the monitoring of requirements evolution. On the other hand the identification of a stable taxonomy may require experience across several projects.

Ageing Requirements Maturity. Any simple account of requirements maturity may be misleading. Our experience shows that any estimation (e.g., by the Requirements Maturity Index) of requirements maturity should be carefully evaluated in the specific context. Any measurement of requirements maturity should be interpreted against the specific development process (e.g., data collection activities, certification constraints, changes classifications, etc.) and management policy (e.g., priorisation of requirements changes, certification constraints, allocation of requirements changes, etc.). Requirements maturity should furthermore take account of ageing factors for requirements. For instance, the elapsed time since requirements were introduced (or modified) may be useful to refine any assessment of requirements maturity. This type of refinement allows us to link the requirements process with the development process. On the other hand the concept of *maturity* can be misleading and misunderstanding for requirements. Future research should further investigate how to distinguish diverse requirements maturities. Further investigations should address the relation between stability, volatility and maturity.

Functional Requirements Evolution. The empirical investigation of the case study points out that it is possible to identify change-prone requirements from a functional viewpoint. The analysis identifies stable and volatile requirements. Moreover it emphasises different distributions of requirements changes for each function. This information may be useful in order to identify reusable requirements (e.g., stable requirements of the system architecture) as well as to devise product-lines ranging around specific variability points. The different distributions of requirements changes highlight dependencies between functional requirements. Requirements dependencies may be useful to refine the assessment of the impact of changes. On the other hand requirements dependencies may be further refined through subsequent releases and similar projects (within the same product-line).

Requirements Evolution Processes. The case study points out our inability to visualise requirements evolution. Our preliminary attempts of visualising requirements evolution stresses that any visual representation should take into account evolutionary process features (e.g., releases, activities, etc.) as well as product features (e.g., types of requirements changes). The representation of requirements evolution processes may allow us to identify similarities between processes and to distinguish them with respect to their complexity. The visualisation of requirements evolution and the identification (by sequence analysis) of different requirements processes show that it is possible to identify different requirements process for each function. This provides us new insights to investigate requirements evolution (processes) in future research.

9.1.2 Smart Card Case Study

The viewpoint analysis highlights issues in the requirement engineering practice drawn from the smart card case study. In spite of sparse data, the analysis effectively points out many requirements evolution aspects that characterise live software production environments. Although changes affect several viewpoints (or management levels) and increase project risk, they are part of learning and understanding processes in software production. From the analysis it is evident how even a single change affects many different socio-technical aspects.

Requirements Evolution Viewpoints. The analysis identifies three different hierarchical viewpoints named *Business*, *Process* and *Product* viewpoints. Each viewpoint corresponds to different processes and requirements within the organisation. For instance, management processes easily deal with high-level system requirements, although they provide limited support for low-level software requirements. This points out struggles with requirements engineering practice. Although viewpoint discrepancies often cause requirements issues (e.g., inconsistency, incorrectness, etc.), all viewpoints provide as a whole a hierarchical management structure that deals with requirements changes. Interestingly, each viewpoint differently perceives requirements evolution. Thus, on one hand viewpoint interactions (hence, stakeholder interactions) give rise to requirements evolution, on the other hand viewpoint interactions represent a mechanism to capture and take into account requirements changes.

Viewpoint Management Support. Each viewpoint seeks different management support. Although process-oriented methodologies allow the planning of project activities, they usually provide limited support to tailor processes to product features. This often requires a shift from process to product-oriented software management. On one hand management processes keep track of requirements changes, on the other hand quantitative approaches (e.g., software metrics) should take into account product as well as environmental aspects. This allows the identification of reusable (product-line) functions. It moreover would be possible to define repeatable processes to allocate low-level software functions to high-level system requirements. There are usually two opposing processes. The first one (top-down) splits and refines requirements. This creates an information flow expansion throughout the development process. The second one (bottom-up) allocates (according to past experience) specific low-level software functions to high-level system requirements. The gap between these two processes represents the extent to which an organisation is able to identify an optimal and effective set of (reusable) software functions. The smaller the gap, the better the ability in reusing low-level software functions and identifying high-level system requirements. Although each viewpoint deals with different requirements, all viewpoints seek support to deal with requirements changes.

Requirements Issues. The structured interviews, using the requirements engineering questionnaire, effectively highlight common requirements issues in live software production environments. Viewpoint divergences clearly point out the different understanding of the requirements engineering practice within the organisation. Unsurprisingly, requirements evolution finds little agreement among the different viewpoints. Although process-oriented management properly capture requirements changes, it provides limited support to development activities. Moreover hierarchical organisation often struggles to communicate requirement changes through each management level (or viewpoint). On the other hand requirements engineering practice provides limited support to communicate specific types of requirements changes. Another major issue is the identification of system boundaries. Although a holistic viewpoint captures many aspects of socio-technical systems, viewpoints differently understand (software) systems. Each viewpoint captures different system boundaries. For instance, a project management level easily identifies high-level system requirements, although these under-specify low-level software interfaces. Software, as subtle part of socio-technical systems, has limited visibility from high-level holistic viewpoints. All these issues highlight concerns for the requirements engineering practice.

9.2 Heterogeneous Requirements Engineering

Heterogeneous engineering provides a comprehensive account of system requirements. Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings. The formal extension of solution space transformation defines a framework to model and capture requirements evolution. The framework relies on a heterogeneous account of requirements. The resulting framework is sufficient to interpret requirements changes. The formal framework captures how requirements evolve through subsequent releases. Hence, it is possible to define requirements evolution in terms of sequential solution space transformations. Intuitively, requirements evolution identifies a path that browses solution spaces.

9.2.1 Heterogeneous Modelling of Requirements Evolution

Heterogeneous engineering considers system production as a whole. It provides a comprehensive account that stresses a holistic viewpoint, which allows us to understand the underlying mechanisms of evolution of socio-technical systems. Heterogeneous engineering is therefore convenient further to understand requirements processes. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings.

The formal extension of solution space transformation, a heterogeneous account of requirements, provides a framework to model and capture requirements evolution. The resulting framework is sufficient to interpret requirements changes. The formal framework captures how requirements evolve through consecutive solution space transformations. Hence, it is possible to define requirements evolution in terms of sequential solution space transformations. The characterisation of requirements and requirements changes allows the definition of requirements evolution. Requirements evolution consists of the requirements specification evolution and the requirements changes evolution. Hence, requirements evolution is a co-evolutionary process. *Heterogeneous Requirements Evolution* gives rise to new insights in requirements engineering.

A New Role for Requirements. Heterogeneous engineering stresses a different role for requirements. The shift from the paradigm of problems searching for solutions (i.e., *problem* $\rightarrow$ *solution*) to the one of solutions searching for problems (i.e., *solution* $\rightarrow$ *problem* $\rightarrow$ *solution*) points out a new role for requirements with respect to (design) solutions and problems. Most software design processes and organisations rely on the first paradigm (i.e., problems searching for solutions). In this case, requirements represent problems to be solved by design solutions. Thus, software production takes into account a certain relationship between requirements, design and system implementation. This relationship implies a specific order to software production phases (e.g., first the collection of system requirements, then the design of solutions and finally the implementation, testing and so on). Regardless the adopted development process, each software production complies with the paradigm of problems searching for solutions. This is one of the reasons because most software development processes start with a requirement phase.

In contrast, heterogeneous engineering takes into account the second paradigm (i.e., solutions searching for problems). Heterogeneous engineering therefore points out that requirements link (design) solutions and given problems observed (by coding, testing, usage, etc.) in the system implementation. Requirements map solutions and problems. This implies a different role for requirements with respect to solutions and problems. On the one hand requirements map solutions to observed problems. On the other hand requirements narrow and browse solution spaces in order to address observed problems. The heterogeneous requirements role highlights new insights in the production of software systems.

Moreover, heterogeneous engineering helps us further to understand the mechanisms of requirements evolution. The modelling of requirements evolution highlights how requirements evolve due to the social shaping of socio-technical systems. On one hand the modelling supports the analysis of evolutionary phenomena (e.g., like in the avionics case study, stability, volatility, dependencies, etc.) in requirements, on the other hand the modelling supports the analysis of stakeholder interactions (e.g., like in the smart card case study, requirements viewpoints) in software production.

Implications for Requirements Processes and Tools. Heterogeneous engineering relies on a different paradigm. Heterogeneous engineering therefore highlights a new role for requirements (engineering) with respect to design solutions and observed system problems. This heterogeneous role has some implications for requirements processes as well as tools, in general, for software production.

Software production usually consists of the process of searching (or designing) solutions to given problems (or requirements). This implies that the requirements process has to search (or elicit) all system requirements in order to find the most suitable solution (by narrowing the solution space). System testing and verification therefore have to provide arguments that support system implementation, design and requirements. Therefore, in practice, verification and testing have to validate solutions by searching problems. In contrast, heterogeneous engineering highlights a new role for requirements. On the one hand the requirements process consists of matching solutions to observed problems. On the other hand the requirements process is to narrow and browse the solution space in order to address observed problems. Therefore, system testing and verification are to reveal problems that will be eventually matched to specific solutions by requirements.

The new role of requirements, with respect to design and problems, points out new scenarios of use for requirements engineering tools. Most requirements engineering tools support the maintenance of traceability between different software deliverables (e.g., requirements, change requests, rationale, design, etc.). On the other hand future requirements engineering tools should also support the mapping of solutions to observed problems. That is, requirements engineering tools should support the analysis of observed problems in order to narrow the solution space. Thus, requirements engineering tools assume a major role in the analysis of observed problems in live production environments.

In summary, the formal framework allows the modelling and gathering of requirements evolution. The framework relies on a heterogeneous account of requirements. Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. Requirements, as mappings between socio-technical solutions and problems, represent an account of the history of socio-technical issues arising and being solved within industrial settings. The formal extension of solution space transformation defines a framework to model and capture requirements evolution. The resulting framework is sufficient to interpret requirements changes. The formal framework captures how requirements evolve through subsequent releases. Hence, it is possible to define requirements evolution in terms of sequential solution space transformations. The characterisation of requirements changes allows the definition of requirements evolution. Requirements evolution consists of the requirements specification evolution and the requirements changes evolution. Hence, requirements evolution is a co-evolutionary process.

9.2.2 Capturing Evolutionary Requirements Dependencies

The formally augmented solution space transformation can capture emergent evolutionary dependency. The empirical analysis of the avionics case study highlights instances of evolutionary dependencies. The analysis points out three different basic dependencies: cascade, self-loop and refinement-loop dependency. The examples show that it is possible to capture evolutionary structures in terms of consecutive solution space transformations. Consecutive solution space transformations identify the history of problems arising and being solved. The underlying formal framework moreover allows the modelling of evolutionary dependency. This supports model-oriented software production.

The modelling of evolutionary dependency highlights that the formal extension of the solution space transformation enables the gathering of evolutionary information at different abstraction levels. Hence, the solution space transformation allows the modelling of different hierarchical features of requirements evolution. This supports related requirements engineering approaches that rely on hierarchical refinements of requirements. The definition of hierarchies of requirements allows the reasoning at different level of abstractions. Unfortunately, requirements changes affect high-level as well as low-level requirements. Moreover, requirements changes often propagate through different requirements levels. Hence, the solution space transformation allows the reasoning of ripple effects of requirements changes at different abstraction levels. With respect to requirements hierarchies, the solution space transformation takes into account anomalies that relate to a lower level of abstraction. For instance, the solution space transformations, this chapter shows, allow the modelling of evolutionary requirements dependencies at the functional level. Although the problem spaces take into account requirements changes due to requirements refinements as well as anomalies at the physical level (e.g., coding and usage feedback).

In practice, the modelling of evolutionary requirements dependency and requirements evolution allows the reconciliation of solutions with observed anomalies. For instance, it would be possible to enhance the reasoning of evolutionary features of requirements, hence requirements evolution. Although most requirements engineering tools support the gathering of requirements (e.g., requirements management tools) and requirements changes (e.g., change management tools), they provide limited support in order to reasoning on observed evolutionary information. Hence, it is difficult to analyse and monitor emergent evolutionary features of requirements. Most requirements methodologies assess the impact of changes using traceability information. Unfortunately, changes affect traceability too. In contrast, the formal extension of solution space transformation allows the modelling of evolutionary requirements dependency, as mappings between dependency models and problems. This represents an account of the history of socio-technical issues arising and being solved within requirements hierarchies.

In summary, the formally augmented solution space transformation allows the gathering of emergent evolutionary features of solutions. Hence, requirements evolution provides the basic mechanisms to capture emergent evolutionary software production.

9.2.3 Towards Requirements Evolution Engineering

Requirements engineering research and practice highlight evolution as an important aspect of software production. Although research results and experience are still patchy, requirements evolution emerges as a comprehensive viewpoint that allows us further to understand the mechanisms of socio-technical system evolution. On the other hand requirements evolution provides new insights in research and practice in software production.

Iterative development processes emphasise evolutionary aspects of software production. Although management and development processes provide overall organisation in terms of development activities and phases, they require to be tailored for specific software systems and design contexts. Empirical analyses allow us to understand evolutionary aspects of software production. Requirements evolution provides new grounds for understanding the mechanisms of socio-technical system evolution. Modelling requirements (evolution) captures the understanding of software system evolution. Although requirements evolution emerges as an important aspect of software production, requirements engineering tools provide limited support for the analysis of requirements evolution.

In contrast, this thesis takes requirements evolution as inherent in software production. It investigates the current understanding of requirements evolution and explores new directions in requirements evolution research. The empirical analysis of industrial case studies highlights software requirements evolution as an important issue. Unfortunately, traditional requirements engineering methodologies provide limited support to capture requirements evolution. This thesis highlights a set of methodologies towards *Requirements Evolution Engineering* (REE). This thesis addresses the problem of empirically understanding and modelling requirements evolution. The empirical analyses of industrial case studies highlight several aspects of requirements evolution. For instance, it is possible to classify requirements according to their volatility and origins. Although empirical analyses successfully capture requirements evolution, they are context sensitive. That is, it is difficult to generalise results as well as methodologies. Requirements evolution modelling allows the tailoring of development processes and artefacts to development environments. The combination of empirical analyses and requirements evolution models captures environmental features (e.g., work practice, product characterisation, etc.). They identify a convenient requirements engineering practice that continuously provides feedback while software production progresses. Although these methodologies provide a comprehensive account of requirements evolution, requirements engineering practice little exploits them. On the other hand requirements engineering practice needs to identify how requirements evolution supports software production. Requirements evolution engineering involves analysis, modelling and practice of requirements evolution. Requirements evolution therefore identifies strategies and methodologies that support software production.

This thesis describes how to use the requirements evolution modelling in three scenarios of practice: modelling requirements evolution, process calibration and requirements evolution regression. Moreover, it describes how the comprehensive account of heterogeneous requirements evolution supports the refinement of design models. Although design phases intend to identify the most suitable solutions that fulfil the given requirements. In contrast, heterogeneous requirements engineering highlights how design models (that is, solutions) support the observation of requirements evolution. The solution space transformation allows the gathering of requirements (evolution) during design. Although these scenarios are descriptive, they provide an overall understanding of how modelling requirements evolution enhances system production. The scenarios therefore represent a contribution towards requirements evolution engineering.

9.3 Evolution as Dependability

Software evolution represents just one aspect of the evolution of socio-technical systems. This chapter describes a taxonomy of evolution: *Software Evolution*, *Architecture (Design) Evolution*, *Requirements Evolution*, *Socio-technical System Evolution*, *Organisation Evolution*. The taxonomy identifies an evolutionary space, which provides a holistic viewpoint in order to analyse and to understand the evolution of socio-technical systems. The taxonomy points out the different aspects of the evolution of socio-technical systems. The review stresses the relationship between system evolution and dependability. Different models and methodologies take into account to some extent the evolution of socio-technical systems. Unfortunately, these models and methodologies rely on different assumptions of the evolution of socio-technical systems. This can cause misunderstandings and issues about system dependability and evolution.

In summary, the taxonomy of evolution represents a starting point for the analysis of socio-technical systems. It identifies a framework that allows the analysis of how socio-technical systems evolve. Moreover, the taxonomy provides a holistic viewpoint that identifies future directions for research and practice on system evolution. On one hand the resulting framework allows the classification of evolution of computer-based systems, on the other hand the framework supports the analysis of the relationships between the different evolutionary phenomena and dependability. Unfortunately, the collection and analysis of evolutionary data are very difficult activities, because evolutionary information is usually incomplete, distributed, unrelated and vaguely understood. The taxonomy of evolution points out that methodologies rely on different assumptions of system evolution. The dependability analysis with respect to evolution identifies a framework. The engineering hints related to each evolutionary phenomenon may serve as basics in order empirically to acquire a taxonomy of evolution. Future work aims to acquire practical experience using the taxonomy in industrial settings.

9.4 Postscript

This thesis supports the empirical investigation and modelling of requirements evolution, hence *Observational Models of Requirements Evolution*. The results provide new insights in requirements engineering and identify the foundations for requirements evolution engineering. Although researchers and practitioners recognise the importance of requirements evolution, there is still a lot of work to do in order to address requirements evolution. This thesis highlights future research and practice directions in requirements engineering. The results point out how a heterogeneous viewpoint is convenient to analyse and model evolutionary requirements aspects in industrial settings. Thus, the heterogeneous requirements evolution allows the understanding of the mechanisms of socio-technical evolution. However, it would be useful further to integrate requirements evolution with other methodologies and models. Requirements evolution is convenient to integrate either engineering approaches (e.g., design, testing, etc.) or social models (e.g., activity theory, distributed cognition, etc.). Hence, requirements evolution stresses that future research and practice should rely on multidisciplinary approaches. This should also support further evaluations in industrial settings. Although heterogeneous requirements evolution originates from empirical evidence of software production, it needs further experience in different live production environments. A key aspect that allows the evaluation is the integration of requirements evolution modelling in various design and management tools that support production activities. Moreover, this supports the analysis of requirements evolution with respect to dependability. In conclusion, this thesis addresses the problem of empirically understanding and modelling requirements evolution. The results highlight multidisciplinary research directions in requirements evolution.

Appendix A

Requirements Engineering Questionnaire

A.1 Business Requirements Engineering

Requirements Methodology Compliance

Question	N/A	UN	VL	L	A	H	VH
A.1.1 *Have the applicable organisation's policies and procedures been identified?*							
A.1.2 *Do requirements comply with these policies and procedures?*							
A.1.3 *Do you document requirements in accordance with the requirements methodology?*							
A.1.4 *Is the cost/benefit analysis prepared in accordance with the appropriate procedures?*							
A.1.5 *Does the requirements phase meet the intent of the requirements methodology?*							
A.1.6 *Is the requirements phase staffed according to procedures?*							
A.1.7 *Will all the applicable policies, procedures and requirements be in effect at the time the system goes in operation?*							
A.1.8 *Will there be new standards, policies and procedures in effect at the time the system goes operational?*							

Business Tolerance Requirements

Question	N/A	UN	VL	L	A	H	VH
A.1.9 *Have the significant financial fields been identified?*							
A.1.10 *Has responsibility for the accuracy and completeness of each financial field been assigned?*							
A.1.11 *Have the accuracy and completeness risks been identified?*							
A.1.12 *Has the individual responsible for each field stated the required precision for financial accuracy?*							
A.1.13 *Has the accounting cutoff method been determined?*							
A.1.14 *Has a procedure been specified to monitor the accuracy of financial information?*							
A.1.15 *Are rules established on handling inaccurate and incomplete data?*							

Business Performance Requirements

Question	N/A	UN	VL	L	A	H	VH
A.1.16 *Will hardware and software be obtained through competitive bidding?*							
A.1.17 *Have cost-effectiveness criteria been defined?*							
A.1.18 *Do you calculate the cost-effectiveness for an application system in accordance with the procedures?*							
A.1.19 *Are the cost-effectiveness procedures applicable to any application?*							
A.1.20 *Could application characteristics cause the actual cost to vary significantly from the projections?*							
A.1.21 *Are there application characteristics that could cause the benefits to vary significantly from the projected benefits?*							
A.1.22 *Is the expected life of projects reasonable?*							
A.1.23 *Does a design phase schedule exist which identifies tasks, people, budgets and costs?*							
A.1.24 *Have you obtained quality certifications (e.g., ISO 9001, CMM, Prince2, TQM, etc.) for your process?*							
A.1.25 *If your organisation is certified to some standards (e.g., ISO 9001, CMM, Prince2, TQM, etc.), which is the (average) level of compliance with them?*							

A.2 Process Requirements Engineering

Requirements Elicitation

Question	N/A	UN	VL	L	A	H	VH
A.2.1 *Do you carry out a feasibility study before starting a new project?*							
A.2.2 *While eliciting requirements are you sensible to organisational and political factors which influence requirements sources?*							
A.2.3 *Do you use business concerns to drive requirements elicitation?*							
A.2.4 *Do you prototype poorly understood requirements?*							
A.2.5 *Do you use scenarios to elicit requirements?*							
A.2.6 *Do you define operational processes?*							
A.2.7 *Do you reuse requirements from other systems which have been developed in the same application area?*							

Requirements Analysis and Negotiation

Question	N/A	UN	VL	L	A	H	VH
A.2.8 *Do you define system boundaries?*							
A.2.9 *Do you use checklists for requirements analysis?*							
A.2.10 *Do you encourage the use of electronic systems (e.g., e-mail) to support requirements negotiations?*							
A.2.11 *Do you plan for conflicts and conflict resolution?*							
A.2.12 *Do you prioritise requirements?*							
A.2.13 *Do you classify requirements using a multidimensional approach which identifies specific types (e.g., hardware-software, changeable-stable, etc.)?*							
A.2.14 *Do you use interaction matrices to find conflicts and overlaps?*							
A.2.15 *Do you perform any risk analysis on requirements?*							

Requirements Validation

Question	N/A	UN	VL	L	A	H	VH
A.2.16 *Do you check that requirements document meets your standards?*							
A.2.17 *Do you organise formal requirements inspections?*							
A.2.18 *Do you use multi-disciplinary teams to review requirements?*							
A.2.19 *Do you involve external (from the project) reviewers in the validation process?*							
A.2.20 *In order to focus the validation process do you define validation checklists?*							
A.2.21 *Do you use prototyping to animate / demonstrate requirements for validation?*							
A.2.22 *Do you propose requirements test cases?*							
A.2.23 *Do you allow different stakeholders to participate in requirements validation?*							

Requirements Management

Question	N/A	UN	VL	L	A	H	VH
A.2.24 *Do you uniquely identify each requirement?*							
A.2.25 *Do you have defined policies for requirements management?*							
A.2.26 *Do you record requirements traceability from original sources?*							
A.2.27 *Do you define traceability policies?*							
A.2.28 *Do you maintain a traceability manual?*							
A.2.29 *Do you use a database to manage requirements?*							
A.2.30 *Do you define change management policies?*							
A.2.31 *Do you identify global system requirements?*							
A.2.32 *Do you identify volatile requirements?*							
A.2.33 *Do you record rejected requirements?*							
A.2.34 *Do you reuse requirements over different projects?*							

Requirements Evolution/Maintenance

Question	N/A	UN	VL	L	A	H	VH
A.2.35 *Has the expected life of the project been defined?*							
A.2.36 *Has the expected frequency of change been defined?*							
A.2.37 *Has the importance of keeping the system up to date functionally been defined?*							
A.2.38 *Has the importance of keeping the system up to date technologically been defined?*							
A.2.39 *Has it been decided who will perform maintenance on the project?*							
A.2.40 *Are the areas of greatest expected change identified?*							
A.2.41 *Has the method of introducing change during development been identified?*							
A.2.42 *Have provisions been included to properly document the application for maintenance purposes?*							

Requirements Process Deliverables

Question	N/A	UN	VL	L	A	H	VH
A.2.43 *Are the deliverables of the requirements process well identified within your organisation?*							
A.2.44 *Is the task of each deliverable well defined within your organisation?*							
A.2.45 *Are the responsibilities for producing the deliverables well defined within your organisation?*							
A.2.46 *Is the deliverables' schedule well defined within your organisation?*							
A.2.47 *Are the responsibilities for reviewing the deliverables well defined within your organisation?*							
A.2.48 *Are relationships among deliverables well defined within your organisation?*							
A.2.49 *Are requirements used as the basis for developing project plans?*							
A.2.50 *Are requirements used as a basis for design?*							
A.2.51 *Are requirements used as the basis for testing?*							
A.2.52 *Are requirements allocated to the software functions of the product?*							

A.3 Product Requirements Engineering

Requirements Description

Question	N/A	UN	VL	L	A	H	VH
A.3.1 *Do you have standards templates / documents for describing requirements?*							
A.3.2 *Do you have a specific lay out for the requirements document to improve readability?*							
A.3.3 *Do you have guidelines how to write requirements?*							
A.3.4 *Do you produce a summary of the requirements?*							
A.3.5 *Do you make a business case for a system?*							
A.3.6 *Do you have a glossary of specialised terms?*							
A.3.7 *Is the requirements document easy to change?*							
A.3.8 *Do you use diagrams appropriately?*							
A.3.9 *Do you supplement natural language with other descriptions of requirements?*							
A.3.10 *Do you specify requirements quantitatively?*							

System Modelling

Question	N/A	UN	VL	L	A	H	VH
A.3.11 *Do you define the system's operating environment?*							
A.3.12 *Do you develop complementary system models?*							
A.3.13 *Do you model the system's environment?*							
A.3.14 *Do you model the system architecture?*							
A.3.15 *Do you use structured methods for system modelling?*							
A.3.16 *Do you define operational processes to reveal process requirements and requirements constraints?*							
A.3.17 *Do you use a data dictionary?*							
A.3.18 *Do you document the links between stakeholder requirements and system?*							
A.3.19 *Do you specify systems using formal specifications?*							

Functional Requirements

Question	N/A	UN	VL	L	A	H	VH
A.3.20 *Can the data required by the application be collected with the desired degree of reliability?*							
A.3.21 *Can the data be collected within the time period specified?*							
A.3.22 *Have the user requirements been defined in writing?*							
A.3.23 *Are the requirements stated in measurable terms?*							
A.3.24 *Has the project solution addressed the user requirements?*							
A.3.25 *Could test data be developed to test the achievement of the objectives?*							
A.3.26 *Have procedures been specified to evaluate the implemented system to ensure the requirements are achieved?*							
A.3.27 *Do the measurable objectives apply to both the manual and automated segments of the application system?*							

Non Functional Requirements

Question	N/A	UN	VL	L	A	H	VH
A.3.28 *Do you identify non functional requirements (e.g., usability, quality, cognitive workload, etc.) for a system?*							
A.3.29 *Have the user functions been identified?*							
A.3.30 *Have the skill levels of the users been identified?*							
A.3.31 *Have the expected levels of supervision been identified?*							
A.3.32 *Has the time span for user function been defined?*							
A.3.33 *Will the counsel of an industrial psychologist be used in designing user functions?*							
A.3.34 *Have user clerical people been interviewed during the requirements phase to identify their concerns?*							
A.3.35 *Have tradeoffs between computer and people processing been identified?*							
A.3.36 *Have the defined user responsibility been presented to the user personnel for comment?*							

Portability Requirements

Question	N/A	UN	VL	L	A	H	VH
A.3.37 *Are significant hardware changes expected during the life of the project?*							
A.3.38 *Are significant software changes expected during the life of the project?*							
A.3.39 *Will the application system be run in multiple locations?*							
A.3.40 *If an on-line application, will different types of terminal be used?*							
A.3.41 *Is the proposed solution dependent on specific hardware?*							
A.3.42 *Is the proposed solution dependent on specific software?*							
A.3.43 *Will the application be run in other countries?*							
A.3.44 *Have the portability requirements been documented?*							

Systems Interface

Question	N/A	UN	VL	L	A	H	VH
A.3.45 *Have data to be received from other applications been identified?*							
A.3.46 *Have data going to other applications been identified?*							
A.3.47 *Have the reliability of interfaced data been defined?*							
A.3.48 *Has the timing of transmitting data being defined?*							
A.3.49 *Has the timing of data being received been defined?*							
A.3.50 *Has the method of interfacing been defined?*							
A.3.51 *Have the interface requirements been documented?*							
A.3.52 *Have future needs of interfaced systems been taken into account?*							

Requirements Viewpoints

Question	N/A	UN	VL	L	A	H	VH
A.3.53 *Do you identify and consult all likely, sources of requirements, system stakeholders?*							
A.3.54 *Do you look for domain constraints?*							
A.3.55 *Do you collect requirements from multiple viewpoints?*							
A.3.56 *Do you use language simply, consistently and concisely for describing requirements?*							
A.3.57 *Do you record requirements traceability from original sources?*							
A.3.58 *Do you record requirements rationale in order to improve requirements understanding?*							

Product-Line Requirements

Question	N/A	UN	VL	L	A	H	VH
A.3.59 *Do you define safety-critical requirements?*							
A.3.60 *Do you identify and analyse hazards?*							
A.3.61 *Do you derive safety (or security, availability, etc.) requirements from hazard analysis?*							
A.3.62 *Do you cross-check operational and functional requirements against safety (or security, availability, etc.) requirements?*							
A.3.63 *Do you collect incident experience (e.g., by incident reports)?*							
A.3.64 *Do you analyse incident reports?*							
A.3.65 *Are responsibilities (for system safety) well identified within your organisation?*							
A.3.66 *Do you define operational profiles for a system?*							
A.3.67 *Do you develop use cases for a system?*							

Failure Impact Requirements

Question	N/A	UN	VL	L	A	H	VH
A.3.68 *Has the financial loss of an application system failure been defined?*							
A.3.69 *Has the financial liss calculation for a failure been extended to show the loss at different time intervals, such as one hour, eight hours, one day, one week, etc.?*							
A.3.70 *Is the proposed system technology reliable and proven in practice?*							
A.3.71 *Has a decision been made as to whether it is necessary to recover this application in the event of a system failure?*							
A.3.72 *Are alternative processing procedures needed in the vent that the system becomes unoperational?*							
A.3.73 *If alternative processing procedures are needed, have they been specified?*							
A.3.74 *Has a procedure been identified for notifying users in the event of a system failure?*							
A.3.75 *Has the desired percent of up-time for the system been specified?*							

Appendix B

Modal Logic

This chapter introduces the logic used to formalise the heterogeneous requirements engineering framework. The basics consist of well-established results in modal logic [Chagrov and Zakharyaschev, 1997]. Although the theoretical results in modal logic extend over several levels of expressiveness (e.g., intuitionistic, propositional, first-order, etc.), this chapter introduces a simple propositional modal logic. The most popular semantics of modal logic relies on the possible worlds framework, or *Kripke structures*. This allows us to define a notion of validity for modal logic, hence *Kripke models*. A proof system, *Tableaux system*, allows us to develop proofs and counter models [Fitting and Mendelsohn, 1998].

B.1 Propositional Modal Logic

This section briefly introduces the syntax and semantics of proposition modal logic [Fitting and Mendelsohn, 1998].

B.1.1 Syntax

The language of propositional modal logic consists of:

- *propositional letters* (also called propositional variables), P, Q, R, etc., they stand for unanalysed propositions

- *propositional constants*, $\top$ and $\bot$, that represent truth and falsehood respectively
- *propositional binary connectives* $\wedge$ (and), $\vee$ (or), $\rightarrow$ (if, then), $\leftrightarrow$ (if, and only if); it is possible to construct other binary operators from these basic connectives
- *unary operator* $\neg$ (not)
- *modal unary operators* $\Box$ (necessarily) and $\Diamond$ (possibly).

Definition B.1 (Propositional Modal Formulas) *The set of propositional modal formulas is specified by the following rules.*

1. *Every propositional letter is a formula.*
2. *If X is a formula, so is* $\neg X$.
3. *If X and Y are formulas, and* $\circ$ *is a binary connective,* $(X \circ Y)$ *is a formula.*
4. *If X is a formula, so are* $\Box X$ *and* $\Diamond X$.

B.1.2 Semantics

Modal logic allows the formalisation of the intuitions about necessity and possibility. There exist many different representations that describe modal logic. Most of them are equivalent from a theoretical viewpoint. For instance, it is possible to describe modal logic in terms of Hypersets (or simply graphs) [Barwise and Moss, 1996]. This section introduces a semantics for propositional modal logic. The semantics relies on the definition of Kripke frames. Intuitively, the Kripke semantics interpretes modal formulas like worlds that are related each other by an accessibility relationship.

Definition B.2 (Kripke Frame) *A frame consists of two parts: a non-empty set,* $\mathcal{G}$*, whose members are generally called possible worlds, and a binary relation,* $\mathcal{R}$*, on* $\mathcal{G}$*, generally called the accessibility relation. Thus, a Kripke frame is a pair* $\langle \mathcal{G}, \mathcal{R} \rangle$.

Note that although *possible world* is a suggestive terminology, possible worlds are any objects (e.g., numbers, sets or even functions, requirements, set of requirements, etc.) whatsoever in the mathematical treatment of frames.

A simple intuitive interpretation considers Kripke frames as *graphs*. The elements of $\mathcal{G}$, are called worlds or points. The accessibility relation, $\mathcal{R}$, identifies the connections (or edges) in the graph. Barwise and Moss in [Barwise and Moss, 1996] give similar definitions using a set-theoretic (i.e., hypersets or non-well-founded sets) representation, which explicitely represents Kripke frames as graphs. We will generally use Γ, Δ, etc. to denote possible worlds. If Γ and Δ are in the relation $\mathcal{R}$, we will write $\Gamma \mathcal{R} \Delta$, and read this as Δ is *accessible from* Γ (or Δ is an *alternative world* to Γ).

A frame is turned into a modal model by specifying which propositional letters are true at which worlds. The following definition and notation formalise Kripke models.

Definition B.3 (Kripke Model) *A propositional modal model, or model for short, is a triple $\mathcal{M} = \langle \mathcal{G}, \mathcal{R}, \Vdash \rangle$, where $\langle \mathcal{G}, \mathcal{R} \rangle$ is a frame and $\Vdash$ is a relation between possible worlds and propositional letters. If $(\mathcal{M}, \Gamma) \Vdash P$ holds, it means that P is true at the world Γ of the collection $\mathcal{G}$ of possible worlds of the model $\mathcal{M}$. If $(\mathcal{M}, \Gamma) \Vdash P$ does not hold, represented by $(\mathcal{M}, \Gamma) \nVdash P$, it means that P is false at the world Γ of the collection $\mathcal{G}$ of possible worlds of the model $\mathcal{M}$.*

If the model under consideration is unambiguous, we can just write $\Gamma \Vdash P$. The truth-relation $\Vdash$ consists of a mapping from the possible worlds and propositional letters. Dually, it is possible to define a mapping from the propositional letters to set of worlds at which each propositional letter is true.

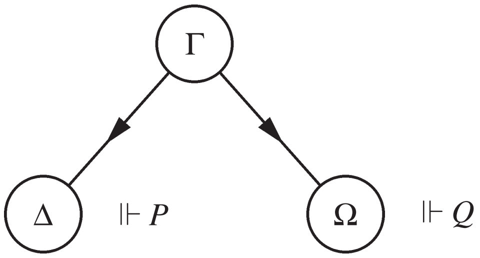
Definition B.4 (Truth in a Model) *Let $\mathcal{M} = \langle \mathcal{G}, \mathcal{R}, \Vdash \rangle$ be a Kripke model. The relation $\Vdash$ is extended (by induction on the construction of the formula) to arbitrary formulas as follows. For each $\Gamma \in \mathcal{G}$,*

$$
\begin{array}{lll}
(\mathcal{M},\Gamma) \Vdash \neg X & \textit{iff} & (\mathcal{M},\Gamma) \nVdash X \\
(\mathcal{M},\Gamma) \Vdash (X \wedge Y) & \textit{iff} & (\mathcal{M},\Gamma) \Vdash X \textit{ and } (\mathcal{M},\Gamma) \Vdash Y \\
(\mathcal{M},\Gamma) \Vdash (X \vee Y) & \textit{iff} & (\mathcal{M},\Gamma) \Vdash X \textit{ or } (\mathcal{M},\Gamma) \Vdash Y \\
(\mathcal{M},\Gamma) \Vdash (X \rightarrow Y) & \textit{iff} & \textit{if } (\mathcal{M},\Gamma) \Vdash X, \textit{ then } (\mathcal{M},\Gamma) \Vdash Y \\
(\mathcal{M},\Gamma) \Vdash (X \leftrightarrow Y) & \textit{iff} & (\mathcal{M},\Gamma) \Vdash X \textit{ if, and only if } (\mathcal{M},\Gamma) \Vdash Y \\
(\mathcal{M},\Gamma) \Vdash \Box X & \textit{iff} & \textit{for every } \Delta \in \mathcal{G}, \textit{ if } \Gamma \mathcal{R} \Delta \textit{ then } (\mathcal{M},\Delta) \Vdash X \\
(\mathcal{M},\Gamma) \Vdash \Diamond X & \textit{iff} & \textit{for some } \Delta \in \mathcal{G}, \Gamma \mathcal{R} \Delta \textit{ and } (\mathcal{M},\Delta) \Vdash X \quad .
\end{array}
$$

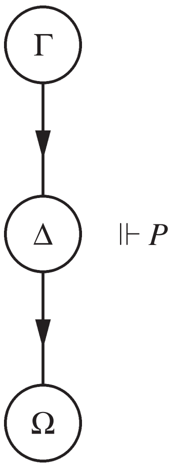
B.1.3 Examples

This section shows some examples that highlight the behaviour of modal models. The representation relies on the similarity between frames and graphs. Models are given using diagrams, with circles representing possible worlds. For two worlds of such a model, Γ and Δ, if $\Gamma \mathcal{R} \Delta$, an arrow from Γ to Δ represents the accessibility relation. The diagrams show explicitely which propositional letters are true at particular worlds. If the diagram does not indicate some propositional letter, it is taken to be false.

Example B.1 *Here is an example of propositional modal models. In this model,* $\Delta \Vdash P \vee Q$*, because* $\Delta \Vdash P$*. Similarly,* $\Omega \Vdash P \vee Q$*. Thus* $\Gamma \Vdash \Box(P \vee Q)$*, because* Δ *and* Ω *are the only possible worlds that are accessible from* Γ*, and* $P \vee Q$ *is true at both of them. On the other hand,* $\Gamma \nVdash \Box P$*, because* $\Omega \nVdash P$ *and* Ω *is accessible from* Γ*. Similarly,* $\Gamma \nVdash \Box Q$ *and* $\Gamma \nVdash \Box P \vee \Box Q$*. Hence,* $\Gamma \nVdash \Box(P \vee Q) \to (\Box P \vee \Box Q)$*.*

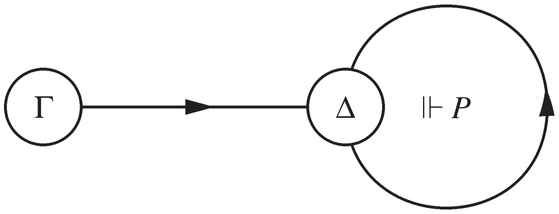


Example B.2 *Let consider the following model. This time,* $\Gamma \nVdash \Box P \to \Box\Box P$*.*



$\Gamma \Vdash \Box P$ since $\Delta \Vdash P$, and Δ is the only world accessible from Γ. If we had $\Gamma \Vdash \Box\Box P$, it would follow that $\Delta \Vdash \Box P$, from which it would follow that $\Omega \Vdash P$, which is not the case. Hence, $\Gamma \nVdash \Box P \to \Box\Box P$.

Example B.3 *This is a counter-example to lots of interesting formulas.*



$\Delta \Vdash \Box P$, because $\Delta \Vdash P$ and Δ is the only world accessible from Δ. But then again, since $\Delta \Vdash \Box P$, it follows that $\Delta \Vdash \Box\Box P$, and similarly $\Delta \Vdash \Box\Box\Box P$, and so on. Similarly, $\Gamma \Vdash \Box P$, because $\Delta \Vdash P$ and Δ is the only world accessible from Γ. But then again, since $\Delta \Vdash \Box P$, it follows that $\Gamma \Vdash \Box\Box P$, and similarly $\Gamma \Vdash \Box\Box\Box P$, and so on. On the other hand, $\Gamma \nVdash P$. Thus at Γ, all the following formulas are false: $\Box P \to P$, $\Box\Box P \to P$, $\Box\Box\Box P \to P$, and so on.

B.1.4 Some Important Logics

Definition B.5 (L-valid) *We say the model $\langle \mathcal{G}, \mathcal{R}, \Vdash \rangle$ is based on the frame $\langle \mathcal{G}, \mathcal{R} \rangle$. A formula X is valid in a model $\langle \mathcal{G}, \mathcal{R}, \Vdash \rangle$, if it it is true at every world of $\mathcal{G}$. A formula X is valid in a frame, if it it is valid in every model based on that frame. Finally, if $\mathbf{L}$ is a collection of frames, X is $\mathbf{L}$-valid if X is valid in every frame in $\mathbf{L}$.*

Definition B.6 *Let $\langle \mathcal{G}, \mathcal{R} \rangle$ be a frame. A frame is:*

1. *reflexive if $\Gamma \mathcal{R} \Gamma$, for every $\Gamma \in \mathcal{G}$*
2. *symmetric if $\Gamma \mathcal{R} \Delta$ implies $\Delta \mathcal{R} \Gamma$, for all $\Gamma, \Delta \in \mathcal{G}$*
3. *transitive if $\Gamma \mathcal{R} \Delta$ and $\Delta \mathcal{R} \Omega$ together imply $\Gamma \mathcal{R} \Omega$, for all $\Gamma, \Delta, \Omega \in \mathcal{G}$*
4. *serial if, for each $\Gamma \in \mathcal{G}$ there is some $\Delta \in \mathcal{G}$ such that $\Gamma \mathcal{R} \Delta$.*

Different modal logics are characterised semantically as the **L**-valid formulas, for particular classes **L** of frames. Table B.1 identifies several frame collections and their corresponding logics. Figure B.1 shows the inclusions among logics.

Table B.1: Some standard modal logics.

Logic	Frame Conditions
K	no conditions
D	serial
T	reflexive
B	reflexive, symmetric
K4	transitive
S4	reflexive, transitive
S5	reflexive, symmetric, transitive

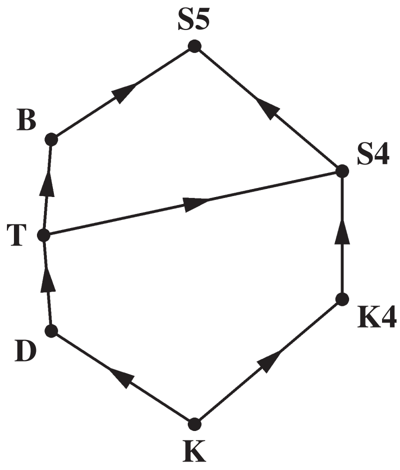


Figure B.1: Inclusions among logics.

B.1.5 Logical Consequence

Definition B.7 (Consequence) *Let* $\mathbf{L}$ *be one of the frame collections given in Table B.1. Also let* S *and* U *be sets of formulas, and let* X *be a single formula.* X *is a consequence in* $\mathbf{L}$ *of* S *as global and* U *as local assumptions, in formulae* $S \models_{\mathbf{L}} U \to X$*, provided: for every frame* $\langle \mathcal{G}, \mathcal{R} \rangle$ *in the collection* $\mathbf{L}$*, for every model* $\langle \mathcal{G}, \mathcal{R}, \Vdash \rangle$ *based on this frame in which all members of* S *are valid, and for every world* $\Gamma \in \mathcal{G}$ *at which all members of* U *are true,* $\Gamma \Vdash X$ *holds.*

Thus $S \models_{\mathbf{L}} U \to X$ means that X is true at those worlds of $\mathbf{L}$ models where the members of U are true, provided the members of S are true throughout the model, i.e., at each world.

B.2 Tableau Proof Systems

This section introduces a Tableau system [Fitting and Mendelsohn, 1998], which uses prefixed formulas. Proofs are particular tableau (tree) of prefixed formulas.

Definition B.8 (Prefix) *A prefix is a finite sequence of positive integers. A prefixed formula is an expression of the form* σX*, where* σ *is a prefix and* X *is a formula.*

Prefixes consist of integers separated by periods, 1.2.3.2.1 for instance. Also, if σ is a prefix and n is a positive integer, $\sigma.n$ is σ followed by a period followed by n. The intuitive idea is that a prefix, σ, names a possible world in some model, and σX tells us that X is true at the world σ names. The aim is that $\sigma.n$ should always name a world that is accessible from the one that σ names.

For any σ, a tableau can be constructed by the extension rules in Table B.2.

Now, for each of the logics in Table B.1, we get a tableau system by adding to the system for $\mathbf{K}$, i.e., Table B.2, the additional rules of B.3 corresponding to each logic.

Definition B.9 (Closure) *A tableau branch is closed if it contains both* $\sigma\ X$ *and* $\sigma\ \neg X$ *for some formula X. A branch that is not closed is open. A tableau is closed if every branch is closed.*

Definition B.10 (Tableau Proof) *A closed tableau for* $1\ \neg Z$ *is a tableau proof of Z, and Z is a theorem if it has a tableau proof.*

Table B.2: Tableau extension rules.

Conjunctive Rules

$$\frac{\sigma\ X \wedge Y}{\begin{array}{c}\sigma\ X\\ \sigma\ Y\end{array}} \qquad \frac{\sigma\ \neg(X \vee Y)}{\begin{array}{c}\sigma\ \neg X\\ \sigma\ \neg Y\end{array}} \qquad \frac{\sigma\ \neg(X \to Y)}{\begin{array}{c}\sigma\ X\\ \sigma\ \neg Y\end{array}} \qquad \frac{\sigma\ X \leftrightarrow Y}{\begin{array}{c}\sigma\ X \to Y\\ \sigma\ Y \to X\end{array}}$$

Disjunctive Rules

$$\frac{\sigma\ X \vee Y}{\sigma\ X \mid \sigma\ Y} \qquad \frac{\sigma\ \neg(X \wedge Y)}{\sigma\ \neg X \mid \sigma\ \neg Y} \qquad \frac{\sigma\ X \to Y}{\sigma\ \neg X \mid \sigma\ Y} \qquad \frac{\sigma\ \neg(X \leftrightarrow Y)}{\sigma\ \neg(X \to Y) \mid \sigma\ \neg(Y \to X)}$$

Double Negation Rule

$$\frac{\sigma\ \neg\neg X}{\sigma\ X}$$

Possibility Rules

If the prefix $\sigma.n$ is new to the branch,

$$\frac{\sigma\ \Diamond X}{\sigma.n\ X} \qquad \frac{\sigma\ \neg\Box X}{\sigma.n\ \neg X}$$

Basic Necessity Rules

If the prefix $\sigma.n$ already occurs on the branch,

$$\frac{\sigma\ \Box X}{\sigma.n\ X} \qquad \frac{\sigma\ \neg\Diamond X}{\sigma.n\ \neg X}$$

Table B.3: Special necessity rules and tableau system for each logic.

The following special necessity rules extend the tableau system for **K**. For every σ and $\sigma.n$ already occurring on the tableau branch:

T $$\frac{\sigma\ \Box X}{\sigma\ X} \qquad \frac{\sigma\ \neg\Diamond X}{\sigma\ \neg X}$$

D $$\frac{\sigma\ \Box X}{\sigma\ \Diamond X} \qquad \frac{\sigma\ \neg\Diamond X}{\sigma\ \neg\Box X}$$

B $$\frac{\sigma.n\ \Box X}{\sigma\ X} \qquad \frac{\sigma.n\ \neg\Diamond X}{\sigma\ \neg X}$$

4 $$\frac{\sigma\ \Box X}{\sigma.n\ \Box X} \qquad \frac{\sigma\ \neg\Diamond X}{\sigma.n\ \neg\Diamond X}$$

4r $$\frac{\sigma.n\ \Box X}{\sigma\ \Box X} \qquad \frac{\sigma.n\ \neg\Diamond X}{\sigma\ \neg\Diamond X}$$

Logic	Rules
D	D
T	T
B	4
K4	B, 4
S4	T, 4
S5	T, 4, 4r

Example B.4 *This example shows a proof for the formula $\Diamond\Box X \to X$. The proof uses the **B** rules. Note that the numbers to the right of each tableau item are for reference only. They are not an official part of the tableau. Formula (1) is the usual starting point. Formulas (2) and (3) are from (1) by a Conjunctive Rule; formula (4) is from (2) by a Possibility Rule; the (5) is from (4) by Special Necessity Rule **B**. Closure is by (3) and (5).*

$$
\begin{array}{lcl}
1 & \neg(\Diamond\Box X \to X) & (1) \\
1 & \Diamond\Box X & (2) \\
1 & \neg X & (3) \\
1.1 & \Box X & (4) \\
1 & X & (5)
\end{array}
$$

Example B.5 *The following is a proof, using the **T** rules, of $[\Box(X \vee Y) \wedge \neg X] \to Y$.*

$$
\begin{array}{lcl}
1 & \neg\{[\Box(X \vee Y) \wedge \neg X] \to Y\} & (1)\ \textit{conjunctive rule} \\
1 & \Box(X \vee Y) \wedge \neg X & (2)\ \textit{conjunctive rule} \\
1 & \neg Y & (3) \\
1 & \Box(X \vee Y) & (4)\ \textit{special necessity rule T} \\
1 & \neg X & (5) \\
1 & X \vee Y & (6)\ \textit{disjunctive rule}
\end{array}
$$

$$
\begin{array}{lll}
1 \quad X \quad (7) & \qquad\qquad & 1 \quad Y \quad (8)
\end{array}
$$

B.2.1 Logical Consequence and Tableaus

Definition B.11 (Assumption Rules) *Let **L** be one of the modal logics for which tableau rules have been given. Also let S and U be sets of formulas. A tableau uses S as global assumptions and U as local assumptions if the following two additional rules are admitted.*

Local Assumption Rule. *If Y is any member of U, then $1\ Y$ can be added to the end of any open branch.*

Global Assumption Rule. *If Y is any member of S, then $\sigma\ Y$ can be added to the end any open branch on which σ appears as a prefix.*

If assumptions, local or global, are involved in a tableau construction for $1\ \neg X$, we refer to the tableau as a derivation of X rather than a proof of X.

B.2.2 Soundness and Completeness

Definition B.12 (Satisfiable) *Suppose S is a set of prefixed formulas. S is satisfiable in the model $\langle \mathcal{G}, \mathcal{R}, \Vdash \rangle$, if there is a way, θ, of assigning to each prefix σ that occurs in S some possible world $\theta(\sigma)$ in $\mathcal{G}$ such that:*

1. *If σ and $\sigma.n$ both occur as prefix in S, then $\theta(\sigma.n)$ is a world accessible from $\theta(\sigma)$, that is, $\theta(\sigma)\,\mathcal{R}\,\theta(\sigma.n)$.*

2. *If σX is in S, then X is true at the world $\theta(\sigma)$, that is, $\theta(\sigma) \Vdash X$.*

A tableau branch is satisfiable if the set of prefixed formulas on it is satisfiable in some model. A tableau is satisfiable if some branch of it is satisfiable.

Proposition B.1 *A closed tableau is not satisfiable.*

Proposition B.2 *If a tableau branch extension rule is applied to a satisfiable tableau, the result is another satisfiable tableau.*

Theorem B.1 (Tableau Soundness) *If X has a tableau proof using the* **K** *rules, X is* **K***-valid.*

Definition B.13 (Saturated) *Let us say a* **K** *tableau is saturated if all appropriate tableau rule applications have been made. More precisely, a tableau is saturated provided, for every branch that is not closed:*

1. *If a prefixed formula other than a possibility or necessity formula occurs on the branch, the applicable rule has been applied to it on the branch.*

2. *If a possibility formula occurs on the branch, the possibility rule has been applied to it on the branch once.*

3. *If a necessity formula occurs on the branch, with prefix σ, the necessity rule has been applied to it on the branch once for each prefix $\sigma.n$ that occurs on the branch.*

If $\mathcal{T}$ is a saturated **K**, and $\mathcal{B}$ is a branch of it that is not closed, we can say several useful things about it. If $\sigma X \wedge Y$ occurs on $\mathcal{B}$, so do both σX and σY, by item 1. Similarly, if $\sigma X \vee Y$ occurs on it, one of σX or σY, again by 1. If $\sigma \Diamond X$ occurs, so will

$\sigma.nX$ for some n. And if $\sigma\Box X$ occurs, so will $\sigma.nX$ for every prefix $\sigma.n$ that occurs on $\mathcal{B}$. Similar remarks apply to other formulas.

Notice that it is always possible to construct a saturated **K** tableau for $1\ \neg X$. To do this, we simply follow a systematic construction procedure. At each pick a branch that has not closed, pick a prefixed formula on it that is not a necessity formula, not atomic, not the negation of an atom, and that has had no rule applied on it on the branch, and do the following. If it is a possibility formula, say $\sigma\Diamond X$, pick the smallest integer n such that $\sigma.n$ does not occur on the branch, and add $\sigma.nX$ to the end of the branch; and then for each necessity rule formula on the branch having σ as prefix, also add the instance having $\sigma.n$ as prefix (so if $\sigma\Box Y$ occurs, add $\sigma.nY$). It is possible to show that this procedure terminates.

Suppose we have a saturate tableau $\mathcal{T}$, and there is a branch $\mathcal{B}$ of it that is not closed. We show how to construct a model in which the branch is satisfiable. Let $\mathcal{G}$ be the collection of prefixes that occur on the branch $\mathcal{B}$. If σ and $\sigma.n$ are both in $\mathcal{G}$, set $\sigma\mathcal{R}\sigma.n$. Finally, if P is a propositional letter, and σP occurs on $\mathcal{B}$, take P to be true at σ, that is, $\sigma \Vdash P$. Otherwise take P to be false. This completely determines a model $\langle\mathcal{G},\mathcal{R},\Vdash\rangle$. Now, the key fact, we need, is that for each formula Z,

if σZ occurs on $\mathcal{B}$ then $\sigma \Vdash Z$

if $\sigma\neg Z$ occurs on $\mathcal{B}$ then $\sigma \nVdash Z$.

The proof of this is by induction on the complexity of the formula Z.

Theorem B.2 (Tableau Completeness) *If X is* **K***-valid, X has a tableau proof using the* **K** *rules.*

Example B.6 *The following is an attempted proof of $\Box(X\wedge\Box Y)\to\Box(\Box X\wedge Y)$ using* *the* **K** *rules.*

1	$\neg(\Box(X \wedge \Box Y) \to \Box(\Box X \wedge Y)$ *using the* **K**)	(1) *conjunctive rule giving (2) and (3)*
1	$\Box(X \wedge \Box Y)$	(2)
1	$\neg(\Box(\Box X \wedge Y))$	(3) *possibility rule giving (4)*
1.1	$\neg(\Box X \wedge Y)$	(4)
1.1	$X \wedge \Box Y$	(5) *conjunctive rule giving (6) and (7)*
1.1	X	(6)
1.1	$\Box Y$	(7)

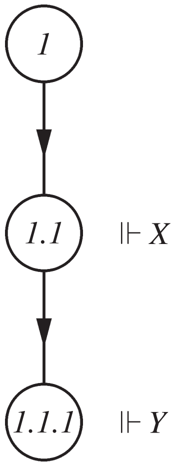
Left branch:

1.1 $\neg\Box X$ (8)
1.1.1 $\neg X$ (10)
1.1.1 Y (11)

Right branch:

1.1 $\neg Y$ (9)

We then continue using the model construction procedure just outlined. Both tableau branches are open. We work with the left one. We construct a model $\langle \mathcal{G}, \mathcal{R}, \Vdash \rangle$ *as follows. Let* $\mathcal{G} = \{1, 1.1, 1.1.1\}$*, the set of prefixes on the left branch. Let* $1 \mathcal{R} 1.1$ *and* $1.1 \mathcal{R} 1.1.1$*. Finally, set* $1.1 \Vdash X$*,* $1.1.1 \Vdash Y$*, and in no other cases are propositional letters true at worlds.*



The prefixed formula $1.1\neg\Box X$ *is on the left branch. And in fact,* $1.1 \not\Vdash \Box X$*, since* $1.1 \mathcal{R} 1.1.1$ *and* $1.1.1 \not\Vdash X$*. Similarly,* $1.1\Box Y$ *occurs on the branch, and* $1.1 \Vdash \Box Y$

since the only possible world of the model that is accessible from 1.1 *is* 1.1.1*, and we have* $1.1.1 \Vdash Y$*. In this way we work our way up the branch, finally verifying that* $\Box(X \wedge \Box Y) \to \Box(\Box X \wedge Y)$ *using the* **K** *is not true at the world* 1*, and hence is not valid.*

Bibliography

[Alexander and Stevens, 2002] Alexander, I. F. and Stevens, R. (2002). *Writing Better Requirements*. Addison-Wesley.

[Anderson and Felici, 2000a] Anderson, S. and Felici, M. (2000a). Controlling requirements evolution: An avionics case study. In Koornneef, F. and van der Meulen, M., editors, *Proceedings of the 19th International Conference on Computer Safety, Reliability and Security, SAFECOMP 2000*, LNCS 1943, pages 361–370, Rotterdam, The Netherlands. Springer-Verlag.

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

[Anderson and Felici, 2001] Anderson, S. and Felici, M. (2001). Requirements evolution: From process to product oriented management. In Bomarius, F. and Komi-Sirviö, S., editors, *Proceedings of the Third International Conference on Product Focused Software Process Improvement, PROFES 2001*, LNCS 2188, pages 27–41, Kaiserslautern, Germany. Springer-Verlag.

[Anderson and Felici, 2002] Anderson, S. and Felici, M. (2002). Quantitative aspects of requirements evolution. In *Proceedings of the Twenty-Sixth Annual International Computer Software and Applications Conference, COMPSAC 2002*, pages 27–32, Oxford, England. IEEE Computer Society.

[Antoniol et al., 1999] Antoniol, G., Canfora, G., and Lucia, A. D. (1999). Estimating the size of changes for evolving object oriented systems: A case study. In *Proccedings of the Sixth International Symposium on Software Metrics, METRICS '99*, pages 250–259, Boca Raton, Florida. IEEE Computer Society.

[Arthur, 1992] Arthur, L. J. (1992). *Rapid Evolutionary Development: Requirements, Prototyping & Software Creation*. John Wiley & Sons.

[Barwise and Moss, 1996] Barwise, J. and Moss, L. (1996). *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. Number 60 in CSLI Lecture Notes. CSLI Publications.

[Bennett et al., 2001] Bennett, S., Skeleton, J., and Lunn, K. (2001). *Schaum's Outline of UML*. Schaum's Outline Series. McGraw-Hill.

[Bergman et al., 2002a] Bergman, M., King, J. L., and Lyytinen, K. (2002a). Large-scale requirements analysis as heterogeneous engineering. *Social Thinking - Software Practice*, pages 357–386.

[Bergman et al., 2002b] Bergman, M., King, J. L., and Lyytinen, K. (2002b). Large-scale requirements analysis revisited: The need for understanding the political ecology of requirements engineering. *Requirements Engineering*, 7(3):152–171.

[Berry and Lawrence, 1998] Berry, D. M. and Lawrence, B. (1998). Requirements engineering. *IEEE Software*, pages 26–29.

[Bijker et al., 1989] Bijker, W. E., Hughes, T. P., and Pinch, T. J., editors (1989). *The Social Construction of Technology Systems: New Directions in the Sociology and History of Technology*. The MIT Press.

[Boehm, 1981] Boehm, B. W. (1981). *Software Engineering Economics*. Prentice-Hall.

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

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

[Boehm et al., 2000] Boehm, B. W. et al. (2000). *Software Cost Estimation with COCOMO II*. Prentice-Hall.

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

[Bowker and Star, 1999] Bowker, G. C. and Star, S. L. (1999). *Sorting Things Out: Classification and Its Consequences*. MIT Press.

[Brooks, 1995] Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, anniversary edition.

[Bustard et al., 2000] Bustard, D., Kawalek, P., and Norris, M., editors (2000). *Systems Modeling for Business Process Improvement*. Artech House.

[CCTA, 1998] CCTA (1998). *Prince2 Manual - Managing successful projects with PRINCE 2*. CCTA.

[Chagrov and Zakharyaschev, 1997] Chagrov, A. and Zakharyaschev, M. (1997). *Modal Logic*. Number 35 in Oxford Logic Guides. Oxford University Press.

[Coakes et al., 2000] Coakes, E., Willis, D., and Lloyd-Jones, R., editors (2000). *The New SocioTech: Graffiti on the Long Wall*. Computer Supported Cooperative Work. Springer-Verlag.

[Coleman et al., 1994] Coleman, D., Ash, D., Lowther, B., and Oman, P. (1994). Using metrics to evaluate software system maintainability. *IEEE Computer*, 27(8):44–49.

[Davis and Hsia, 1994] Davis, A. M. and Hsia, P. (1994). Giving voice to requirements engineering. *IEEE Software*, pages 12–16.

[De Michelis et al., 1998] De Michelis, G., Dubois, E., Jarke, M., Matthes, F., Mylopoulos, J., Schmidt, J. W., Woo, C., and Yu, E. (1998). A three-faceted view of information systems. *Communications of the ACM*, 41(12):64–70.

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

[Fagin et al., 2003] Fagin, R., Halpern, J. Y., Moses, Y., and Vardi, M. Y. (2003). *Reasoning about Knowledge*. The MIT Press.

[Felici, 2000] Felici, M. (2000). Dependability perspectives in requirements engineering. In *Student Forum, Workshops and Abstracts Proccedings of The International Conference on Dependable Systems and Networks, DSN 2000*, pages A43–A45, New York, New York, USA. IEEE Computer Society.

[Felici, 2003] Felici, M. (2003). Taxonomy of evolution and dependability. In *Proceedings of the Second International Workshop on Unanticipated Software Evolution, USE 2003*, pages 95–104, Warsaw, Poland.

[Felici et al., 2000] Felici, M., Pasquini, A., and Sujan, M.-A. (2000). 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.

[Fenton and Pfleeger, 1996] Fenton, N. E. and Pfleeger, S. L. (1996). *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, second edition.

[Fitting and Mendelsohn, 1998] Fitting, M. and Mendelsohn, R. L. (1998). *First-Order Modal Logic*. Kluwer Academic Publishers.

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

[Foote and Yoder, 1996] Foote, B. and Yoder, J. (1996). Evolution, architecture, and metamorphosis. In Vlissides, J. M., Coplien, J. O., and Kerth, N. L., editors, *Pattern Languages of Program Design 2*, chapter 13. Addison-Wesley.

[Gargantini and Heitmeyer, 1999] Gargantini, A. and Heitmeyer, C. (1999). Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'99*, volume 1687 of *LNCS*, pages 146–162. Springer-Verlag.

[Gilb and Graham, 1993] Gilb, T. and Graham, D. (1993). *Software Inspection*. Addison-Wesley.

[Gotel and Finkelstein, 1994] Gotel, O. C. and Finkelstein, A. C. (1994). An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering, ICRE'94*, pages 94–101. IEEE Conputer Society Press.

[Graves et al., 2000] Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. (2000). Predicting fault incident using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661.

[Gunter et al., 2000] Gunter, C. A., Gunter, E. L., Jackson, M., and Zave, P. (2000). A reference model for requirements and specifications. *IEEE Software*, pages 37–43.

[Halpern, 2003] Halpern, J. Y. (2003). *Reasoning about Uncertainty*. The MIT Press.

[Hammer et al., 1998] Hammer, T. F., Huffman, L. L., and Rosenberg, L. H. (1998). Doing requirements right the first time. *CROSSTALK The Journal of Defense Software Engineering*, pages 20–25.

[Harker et al., 1993] Harker, S., Eason, K., and Dobson, J. (1993). 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. IEEE Computer Society Press.

[Heinrich, 1950] Heinrich, H. W. (1950). *Industrial accident prevention: a scientific approach*. McGraw-Hill, 3rd edition.

[Heitmeyer, 2002] Heitmeyer, C. L. (2002). Software cost reduction. In Marciniak, J. J., editor, *Encyclopedia of Software Engineering*. John Waley & Sons, 2nd edition.

[Heitmeyer et al., 1998] Heitmeyer, C. L., Kirby, J., Labaw, B. G., and Bharadwaj, R. (1998). SCR*: A toolset for specifying and analyzing software requirements. In Hu, A. J. and Y.Vardi, M., editors, *Proceedings of the 10th International Conference on Computer Aided Verification, CAV '98*, volume 1427 of *LNCS*, pages 526–531.

[Hitchins, 1992] Hitchins, D. K. (1992). *Putting Systems to Work*. John Wiley & Sons.

[Hoffman and Weiss, 2001] Hoffman, D. M. and Weiss, D. M., editors (2001). *Software Fundamentals: Collected Papers by David L. Parnas*. Addison-Wesley.

[Hooks and Farry, 2001] Hooks, I. F. and Farry, K. A. (2001). *Customer-Centered Products: Creating Successful Products Through Smart Requirements Management*. Amacom.

[Hughes and Hughes, 2000] Hughes, A. C. and Hughes, T. P., editors (2000). *Systems, Experts, and Computers: The Systems Approach in Management and Engineering, World War II and After*. The MIT Press.

[Hull et al., 2002] Hull, E., Jackson, K., and Dick, J. (2002). *Requirements Engineering*. Springer-Verlag.

[Hunt, 2000] Hunt, J. (2000). *The Unified Process for Practitioners: Object Oriented Design, UML andJava*. Practitioner Series. Springer-Verlag.

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

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

[ISO/IEC, 2001] ISO/IEC (2001). *ISO/IEC 9126 - Software engineering - Product quality*. ISO/IEC.

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

[Jarke and Pohl, 1994] Jarke, M. and Pohl, K. (1994). Requirements engineering in 2001: (virtually) managing a changing reality. *Software Engineering Journal*, pages 257–266.

[Jirotka and Goguen, 1994] Jirotka, M. and Goguen, J. A., editors (1994). *Requirements Engineering: Social and Technical Issues*. Computers and People Series. Academic Press.

[Kemerer and Slaughter, 1999] Kemerer, C. F. and Slaughter, S. (1999). An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509.

[Kotonya and Sommerville, 1996] Kotonya, G. and Sommerville, I. (1996). Requirements engineering with viewpoints. *Software Engineering Journal*, 11:5–18.

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

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

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

[Laprie, 1995] Laprie, J.-C. (1995). 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.

[Laprie et al., 1998] Laprie, J.-C. et al. (1998). Dependability handbook. Technical Report LAAS Report no 98-346, LIS LAAS-CNRS.

[Lauesen, 2002] Lauesen, S. (2002). *Software Requirements: Styles and Techniques*. Addison-Wesley.

[Leffingwell and Widrig, 2003] Leffingwell, D. and Widrig, D. (2003). *Managing Software Requirements: A Use Case Approach*. Object Technology Series. Addison-Wesley, second edition.

[Lehman, 1998] Lehman, M. (1998). Software's future: Managing evolution. *IEEE Software*, pages 40–44.

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

[Lehman et al., 1998] Lehman, M., Perry, D., and Ramil, J. (1998). On evidence supporting the feast hypothesis and the laws of software evolution. In *Proceedings of Metrics '98*, Bethesda, Maryland.

[Leveson, 1995] Leveson, N. G. (1995). *SAFEWARE: System Safety and Computers*. Addison-Wesley.

[Leveson, 2000] Leveson, N. G. (2000). Intent specifications: An approach to building human-centered specifications. *IEEE Transactions on Software Engineering*, 26(1):15–35.

[Levine et al., 2000] Levine, F., Locke, C., Searls, D., and Weinberger, D. (2000). *The Cluetrain Manifesto: The end of business as usual*. FT.com. Pearson Education.

[Linscomb, 2003] Linscomb, D. (2003). Requirements engineering maturity in the CMM. *CROSSTALK The Journal of Defence Software Engineering*, 16(12):25–28.

[Littlewood et al., 2001] Littlewood, B., Popov, P., and Strigini, L. (2001). Modelling software design diversity: a review. *ACM Computing Surveys*, 33(2):177–208.

[Littlewood and Strigini, 2000] Littlewood, B. and Strigini, L. (2000). Software reliability and dependability: a roadmap. In Finkelstein, A., editor, *The Future of Software Engineering*, pages 177–188. ACM Press, Limerick.

[Lutz and Mikulski, 2003] Lutz, R. R. and Mikulski, I. C. (2003). Operational anomalies as a cause of safety-critical requirements evolution. *The Journal of Systems and Software*, 65(2):155–161.

[Lyu, 1996] Lyu, M. R., editor (1996). *Handbook of Sofwtare Reliability Engineering*. IEEE Computer Society Press.

[MacKenzie, 1990] MacKenzie, D. A. (1990). *Inventing Accuracy: A Historical Sociology of Nuclear Missile Guidance*. The MIT Press.

[MacKenzie and Wajcman, 1999] MacKenzie, D. A. and Wajcman, J., editors (1999). *The Social Shaping of Technology*. Open University Press, 2nd edition.

[Mens and Galan, 2002] Mens, T. and Galan, G. H. (2002). 4th workshop on object-oriented architectural evolution. In Frohner, A., editor, *Proceedings of the ECOOP 2001 Workshops*, LNCS 2323, pages 150–164. Springer-Verlag.

[Norman, 1998] Norman, D. A. (1998). *The Invisible Computer*. The MIT Press Cambridge, Massachusetts.

[O'Hara et al., 2000] O'Hara, M. T., Kavan, C. B., and Watson, R. T. (2000). Information systems implementation and organisational change: A socio-technical systems approach. In Coakes, E., Willis, D., and Lloyd-Jones, R., editors, *The New SocioTech: Graffiti on the Long Wall*, Computer Supported Cooperative Work, chapter 14. Springer-Verlag.

[Parnas and Madey, 1995] Parnas, D. L. and Madey, J. (1995). Functional documents for computer systems. *Science of Computer Programming*, 25:41–61.

[Paulk et al., 1993] Paulk, M. C. et al. (1993). Key practices of the capability maturity model, version 1.1. Technical Report CMU/SEI-93-025, Software Engineering Institute, Carnegie Mellon University.

[Perrow, 1999] Perrow, C. (1999). *Normal Accidents: Living with High-Risk Technologies*. Princeton University Press.

[Perry, 1994] Perry, D. E. (1994). Dimensions of software evolution. In *Proceedings of the IEEE International Conference on Software Maintenance*. IEEE Computer Society Press.

[Petroski, 1992] Petroski, H. (1992). *To Engineer is Human: The Role of Failure in Successful Design*. Vintage Books.

[Petroski, 1994] Petroski, H. (1994). *Design Paradigms: Case Histories of Error and Judgement in Engineering*. Cambridge University Press.

[Pfleeger, 1998] Pfleeger, S. L. (1998). *Software Engineering: Theory and Practice*. Prentice-Hall.

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

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

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

[Reason, 1997] Reason, J. (1997). *Managing the Risks of Organizational Accidents*. Ashgate Publishing Limited.

[Robertson and Robertson, 1999] Robertson, S. and Robertson, J. (1999). *Mastering the Requirements Process*. Addison-Wesley.

[Rolland, 1994] Rolland, C. (1994). Modeling the evolution of artifacts. In *Proceedings of the First IEEE International Conference on Requirements Engineering, ICRE '94*, pages 216–219.

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

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley.

[Rushby, 2002] Rushby, J. (2002). Using model checking to help to discover mode confusions and other automation surprises. *Reliability Engineering and System Safety*, 75:167–177.

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

[Schmid and Verlage, 2002] Schmid, K. and Verlage, M. (2002). The economic impact of product line adoption and evolution. *IEEE Software*, 19(4):50–57.

[Sha et al., 1995] Sha, L., Rajkumar, R., and Gagliardi, M. (1995). A software architecture for dependable and evolvable industrial computing systems. Technical Report CMU/SEI-95-TR-005, CMU/SEI.

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

[Sommerville, 2001] Sommerville, I. (2001). *Software Engineering*. Addison-Wesley, sixth edition.

[Sommerville and Sawyer, 1997a] Sommerville, I. and Sawyer, P. (1997a). *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons.

[Sommerville and Sawyer, 1997b] Sommerville, I. and Sawyer, P. (1997b). Viewpoints: principles, problems and a practical approach to requirments engineering. *Annals of Software Engineering*, pages 101–130.

[Stark et al., 1998] Stark, G., Skillicorn, A., and Ameele, R. (1998). An examination of the effects of requirements changes on software releases. *CROSSTALK The Journal of Defence Software Engineering*, pages 11–16.

[Stirling, 2001] Stirling, C. (2001). *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer-Verlag.

[Storey, 1996] Storey, N. (1996). *Safety-Critical Computer Systems*. Addison-Wesley.

[Van Buren and Cook, 1998] Van Buren, J. and Cook, D. C. (1998). Experiences in the adoption of requirements engineering technologies. *CROSSTALK The Journal of Defence Software Engineering*, pages 3–10.

[van Lamsweerde, 2000] van Lamsweerde, A. (2000). Requirements engineering in the year 00: A research perspective. In *Proceedings of the 2000 International Conference on Software Engineering, ICSE 2000*, pages 5–19, Limerick, Ireland.

[Vincenti, 1990] Vincenti, W. G. (1990). *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*. The Johns Hopkins University Press.

[Weinberg, 1997] Weinberg, G. M. (1997). *Quality Software Management. Volume 4: Anticipating Change*. Dorset House.

[Weiss and Lai, 1999] Weiss, D. M. and Lai, C. T. R. (1999). *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley.

[Weiss et al., 2003] Weiss, K. A., C.Ong, E., and Leveson, N. G. (2003). Reusable specification components for model-driven development. In *Proceedings of the International Conference on System Engineering, INCOSE 2003*.

[Wiegers, 1999] Wiegers, K. E. (1999). *Software Requirements*. Microsoft Press.

[Wiels and Easterbrook, 1999] Wiels, V. and Easterbrook, S. (1999). Formal modeling of space shuttle software change requests using SCR. In *Proceedings of the Fouth IEEE International Symposium on Requirements Engineering, RE'99*, pages 114–122. IEEE Computer Society.

[Williams and Edge, 1996] Williams, R. and Edge, D. (1996). The social shaping of technology. *Research Policy*, 25(6):865–899.

[Williams et al., 2000] Williams, R., Slack, R., and Stewart, J. (2000). Social learning in multimedia. Final report, EC targeted socio-economic research, project: 4141 PL 951003, Research Centre for Social Sciences, The University of Edinburgh.

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

[Zowghi and Offen, 1997] Zowghi, D. and Offen, R. (1997). 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. IEEE Computer Society Press.