# Chapter 3

# Structuring evolution: on the evolution of socio-technical systems

Massimo Felici

The University of Edinburgh

## 1 Introduction

In order to understand socio-technical (or computer-based) systems, it is important to understand the role of the environment(s) in which these systems are developed and deployed. Socio-technical systems are open, as opposed to closed, with respect to their surroundings. The interactions between socio-technical systems and their environments (which often involve other socio-technical systems) highlight the *Social Shaping of Technology* (SST) [32; 43]. Although this comprehensive understanding allows us to characterise socio-technical systems (with respect to their environments), it provides limited support to understand the mechanisms supporting *sustainable* socio-technical systems. This is due to the lack of methodologies addressing the evolution of socio-technical systems.

Technology driven methodologies often rely on the strict configuration management of socio-technical systems, although it inhibits the evolution of socio-technical systems. Therefore, *evolution* provides a convenient viewpoint for looking at socio-technical systems. Evolution of socio-technical systems is a desirable feature, because it captures emerging social needs. Moreover, evolution allows, if properly understood and managed, the mitigation of socio-technical failures. Unfortunately, current methodologies provide limited support with respect to the evolution of socio-technical systems.

This work analyses mechanisms and examples of evolution of socio-technical structures (e.g. architecture, traceability, coupling, dependency, etc.), hence socio-technical systems. On the one hand, evolution may affect structures. On the other hand, structures may support evolution too. Modelling (requirements) evolution [14] captures how (design) structures evolve due to stakeholder interaction. Heterogeneous engineering provides a comprehensive account of system requirements. *Successfully inventing the technology, turned out to be heterogeneous engineering, the engineering of the social as well as the physical world* [31]. Heterogeneous engineering stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolu-

tion 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 [6; 7]. The formal extension of a heterogeneous account of requirements provides a framework to model and capture requirements evolution [14]. The application of the proposed framework provides further evidence that it is possible to capture and model evolutionary information about requirements [14]. Finally, the identification of a broad spectrum of evolutions in socio-technical systems points out strong contingencies between system evolution and dependability. We argue that the better our understanding of socio-technical evolution, the better system dependability.

## 2 A taxonomy of evolution

Heterogeneous engineering [31] stresses a holistic viewpoint that allows us to understand the underlying mechanisms of evolution of socio-technical systems. Although evolution is a necessary feature of socio-technical systems, it often increases the risk of failures. This section introduces 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 highlights strong contingencies between system evolution and dependability. 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 [44]. Social learning involves the process of fitting technological artefacts into existing socio-technical systems (i.e. heterogeneous networks of machines, systems, routines and culture) [44].

*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* [20] evolution identifies the evolution of technological artefacts (e.g. hardware and software). Whereas, *soft* [20] 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.

Figure 1 shows the evolutionary space. 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 [44]. The evolutionary space supports the analysis of evolution of socio-technical systems.

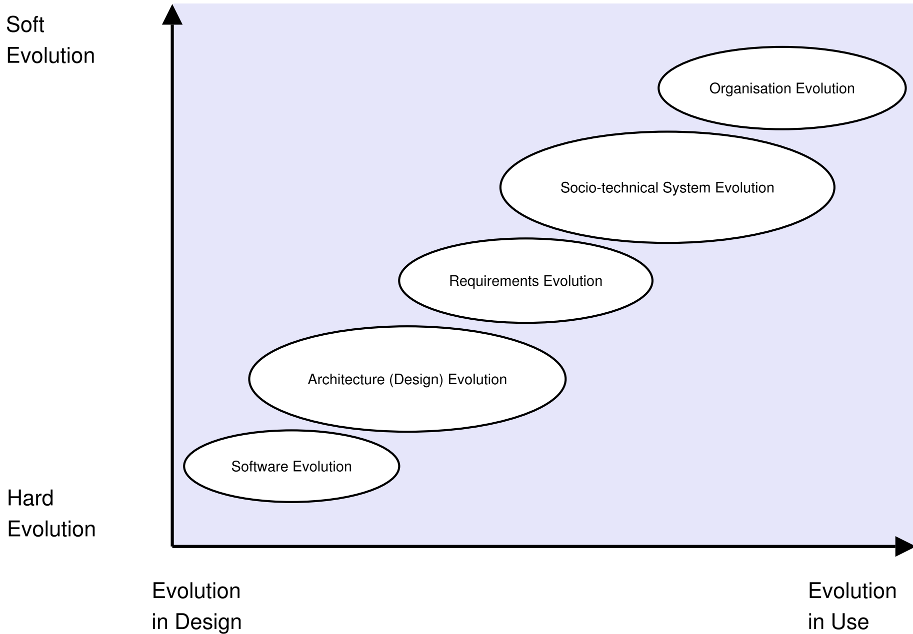


**Fig. 1.** Evolutionary space for socio-technical systems

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 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. Note 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. [17; 34]) that categorise and structure engineering aspects of socio-technical systems.

## 3 Heterogeneous evolution modelling

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 [41]. Although requirements serve as a 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 involves both the system approach [21] as well as the social shaping of technology [32]. 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 [6; 7].

This section introduces a formal framework to model and capture requirements evolution [14]. The framework relies on a heterogeneous account of requirements. Heterogeneous engineering provides a comprehensive account of system requirements. The underlying hypothesis is that heterogeneous requirements engineering is sufficient to capture requirements 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 [6; 7]. The formal extension of solution space transformation defines a framework to model and capture requirements evolution [14]. 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. The remainder of this section briefly summarises the formal extension of solution space transformation (see [14] for an extensive introduction of the formal framework).

## 3.1 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 [8] further explains the complex socio-technical interactions that occur during system production. Requirements are socially shaped (that is, constructed and negotiated) [32] through sequences of mappings between solution spaces and problem spaces [6; 7]. These mappings identify a *Functional Ecology* model that defines requirements as emerging from solution space transformations [6; 7]. This implies that requirements engineering processes consist of solutions searching for problems, rather than the other way around (that is, problems searching for solutions). This heterogeneous account of requirements is convenient to capture requirements evolution. This section describes a formal extension of the solution space transformation [14].

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[1] [10; 16]. Intuitively, a solution space is just a collection of solutions, which represent the organisational knowledge acquired by the social shaping of technical systems. Solutions 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 and problems are Kripke models and formulae of (propositional) modal logic, respectively. 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). Based on the syntax of Kripke models, proof systems (e.g. *Tableau systems*) consist of procedural rules[2] (i.e. inference rules) that allow us to prove formulae's validity or to find counterexamples (or countermodels).

### Solution space

Technical solutions represent organisational knowledge that may be available or unavailable at a particular time according to environmental constraints. A *Local Solution*

[1]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. [39]) bases (i.e. syntax, semantics and proof systems). The definitions still remain sound and valid due to construction arguments.

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

*Space* is the collection of available (or believed available) solutions in an organisation. 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 [6]. 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.

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.

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 us 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

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

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 $\Delta$ is an alternative (or possible) world for $\Gamma$. For every world $\Gamma$, 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 $\Gamma$, if $\varphi$ is true in all the worlds accessible from $\Gamma$. Whereas, $\Diamond\varphi$ is true in $\Gamma$, if $\varphi$ is true at least in one world $\Delta$ 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.

### 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 a hierarchy of problem spaces.

An *anomaly* [31] 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. 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 [16] 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 [13] and uncertainty [18]. Modalities therefore capture problems highlighted by stakeholders and allow us to reason about 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 [6; 7]. As opposed to solutions pointing to problems, *Problem Contextualisation* is the mapping of problems to solutions.

### 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 mapping 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.

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*.

### 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

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 solves) the selected problems. The resolution of the selected problems identifies the proposed future solutions.

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 (e.g. a Tableau system for propositional modal logic). If a formula is satisfiable (that is, there exists 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.

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*.

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

### 3.2 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 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 affects 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.

The requirements specification $\mathbf{RS}^t$ (i.e. the mappings $\mathbf{R}_o^t$ and $\mathbf{R}_c^t$) 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 $\mathbf{R}_o^t$ or in $\mathbf{R}_c^t$, 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 $\mathbf{R}_o^t$ and $\mathbf{R}_c^t$) 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 changing requirements that are likely to ripple cascade effects [6; 7].

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 time 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 during 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.

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[3].

### 3.3 Requirements evolution

The solution space transformation captures requirements as mappings between solutions and problems. Requirements, as mappings between socio-technical solutions and

[3]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 $A - B = \{a \mid a \in A \text{ and } a \notin B\}$. The *union* of $A$ and $B$ is the set $A \cup B = \{a \mid a \in A \text{ or } a \in B\}$.

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

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).

## 4 Capturing evolutionary dependencies

Requirements management methodologies and practices rely on requirements traceability [38]. Although requirements traceability provides useful information about requirements, traceability manifests emergent evolutionary aspects just as requirements do. Realistically, it is 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 (e.g. scalability, evolution and timeliness) that affect traceability practice. For instance, traceability provides limited support to capture indirect emerging dependencies [27]. Requirements changes may trigger subsequent changes in requirements [27]. This results in a cascade effect of requirements changes. Thus, requirements dependencies emerge due to requirements changes. Traceability has therefore to reflect emerging dependencies. 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.

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* [23]. 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, dependent on them [38]. 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* [22] and *Evolutionary Dependency* [14]. Rich Traceability

[22] 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. Note that rich traceability gives rise to hierarchical refinements of requirements. This is similar to Intent Specifications [27], 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 levels of abstractions [27]. Unfortunately, requirements changes affect high-level as well as low-level requirements in Intent Specifications. Moreover, requirements changes often propagate through different requirements levels [27]. Hence, it is very difficult to monitor and control the multi-level cascade effect of requirements changes. In accordance with the notion of semantic coupling, Intent Specifications support strategies to reduce the cascade effect of changes [42]. Although these strategies support the analysis and design of evolving systems, they provide limited support to understand the evolution of high-level system requirements.

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. *Evolutionary Dependency* identifies how changes eventually propagate through emergent requirements dependencies. Requirements dependencies, as an instance of traceability, identify relationships between requirements and constrain software production. Moreover, requirements dependencies constrain requirements evolution. Thus, it is important to capture these dependencies in order further to understand requirements evolution. 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. Changes in rationale can trigger subsequent requirements changes. Requirements' responses to changed 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, sub-

system 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 feed back (or refine) requirements through (circular) requirements dependencies. The remainder of this section shows how the formally augmented solution space transformation captures evolutionary requirements dependencies. Examples drawn from an avionics case study provide a realistic instance of requirements dependencies [1; 2; 3; 4; 14]. These examples show how the heterogeneous framework captures evolutionary features of requirements, hence requirements evolution.

### 4.1 Modelling dependencies

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.

The empirical analysis of an avionics case study points out several instances of requirements dependencies [1; 2; 3; 4; 14]. 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.

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. 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 1 (Cascade Dependency).* This is an instance of the cascade effect of requirements changes. It captures the fact that changes in some requirements trigger changes

[4]Note that the Kripke models in the examples throughout this section 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.

into other requirements eventually. Figure 2 shows a Kripke model of the cascade dependency between two functions, i.e. F1 and F2. 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.

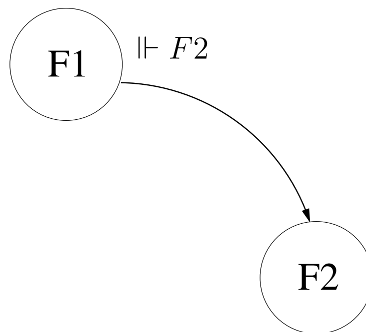


**Fig. 2.** A Kripke model of the evolutionary dependency between F1 and F2

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.

Note that the evolutionary dependency graphs (models) 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 [27]. This complies with the features that other requirements engineering models highlight (e.g. [22; 27]). 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 [27].

*Example 2 (Self-loop and Refinement-loop Dependencies).* Self-loop dependencies identify self-dependencies, that is, some requirements depend on themselves. This dependency implies that some changes require subsequent related refinements of requirements. Refinement-loop dependencies, similarly, identify 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 stakeholders negotiate or mediate requirements through subsequent refinements.

It is possible to extend the dependency graphs for self-loop and refinement-loop dependencies to Kripke models. Figure 3 shows a Kripke model that represents a self-loop dependency. Any reflexive Kripke model captures self-loop dependencies. On the other hand in any reflexive model each possible world has a reflexive loop [5]. Figure 4 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. Note that, from a logic viewpoint, the two Kripke frames (see Figure 3 and 4) are bisimilar in the theory of non-well founded sets (or Hypersets) [5].

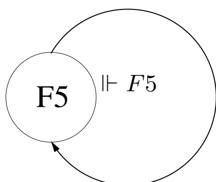


**Fig. 3.** A Kripke model of the self-loop dependency for F5

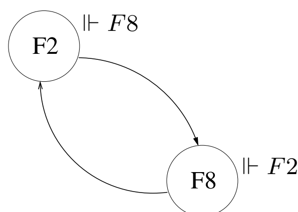


**Fig. 4.** A Kripke model of the refinement-loop dependency between F2 and F8

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.

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 allows the capture of complex ones. This results in the combination (or composition) of the underlying models (e.g. *Generation*, *Reduction* and *Disjoint Union* are three very important operations on modal logic models and frames which preserve truth and validity [10]).

*Example 3 (Complex Dependencies).* Figure 5 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.

### 4.2 Capturing emergent dependencies

The formally augmented solution space transformation captures emergent evolutionary dependencies. That is, it is possible to capture how evolutionary dependencies change

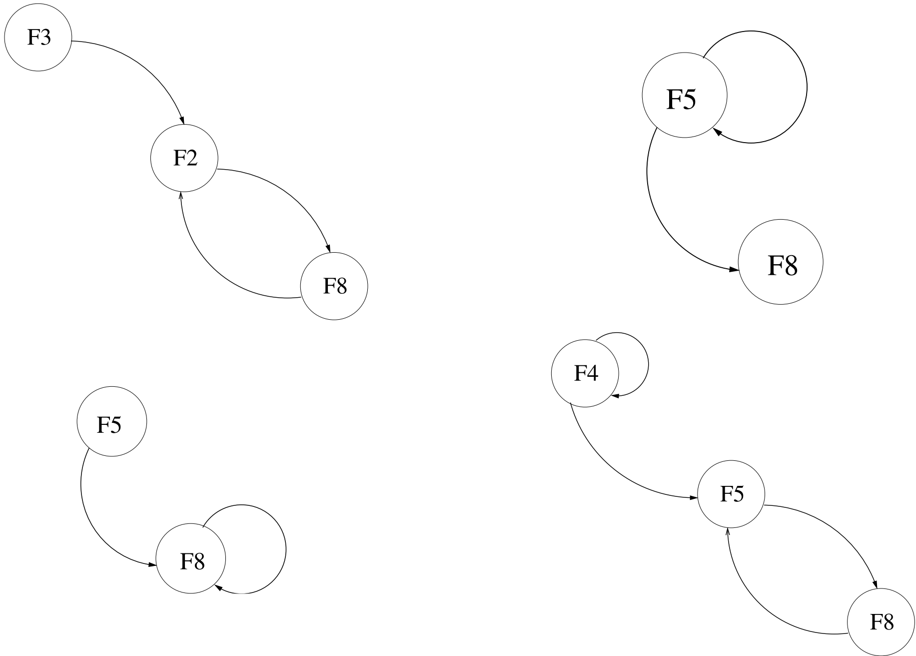


**Fig. 5.** Examples of complex evolutionary dependencies

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 4 (Cascade Dependency continued).* Let us 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 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 in F2". It is easy to see that any disconnected Kripke frame 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 a model exists that satisfies the given problem. This means to prove the validity of $\neg(\Box F2 \rightarrow \Diamond F1)$. 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 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 highlighted by anomaly reports.

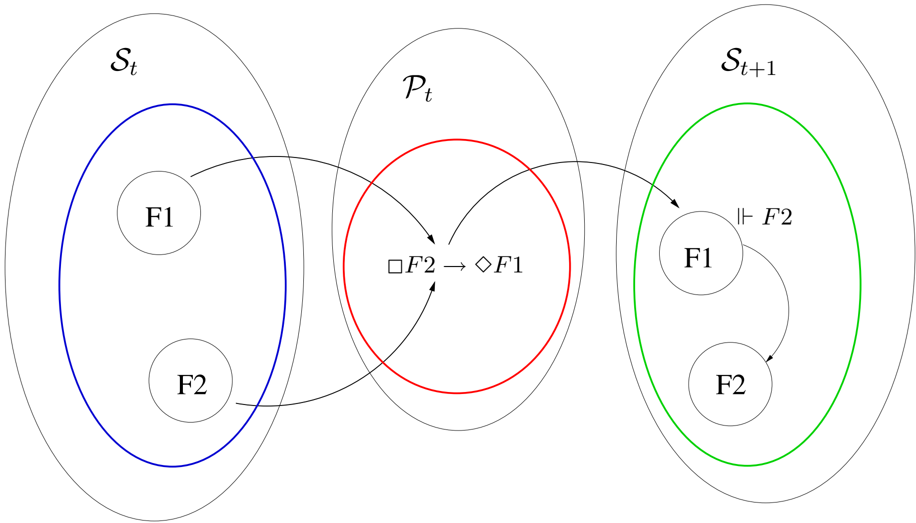


**Fig. 6.** A solution space transformation for F1 and F2

A future solution space transformation can also capture the evolutionary dependency between F2 and F8. Assume that a refinement-loop dependency exists 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 7 shows a Kripke frame modelling the dependency between F1, F2 and F8. It represents a proposed solution. Similarly, consecutive solution space transformations can capture the evolutionary dependency for all the system functions.

The evolutionary dependency models allow the gathering of engineering information. On the one hand the models capture the history of socio-technical issues arising

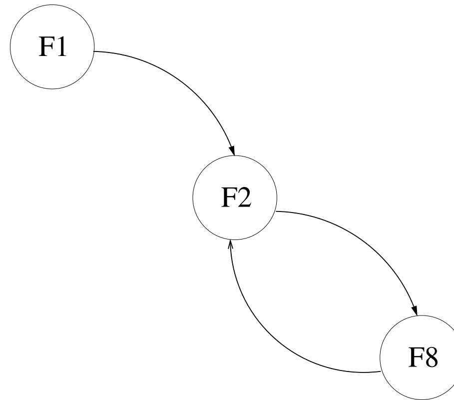


**Fig. 7.** A Kripke frame that captures the dependency between F1, F2 and F8

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 change. Moreover, information about requirements evolution and volatility would allow the adjustment of cost models [9]. 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 high cost (e.g. infinite or non-affordable cost).

*Example 5.* Figure 8 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 with 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 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.

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

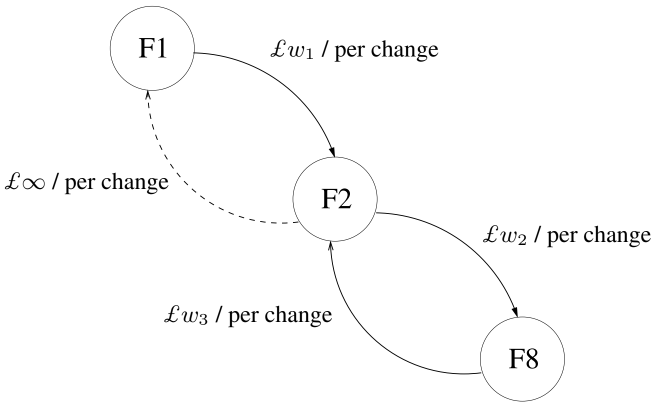


**Fig. 8.** A weighted model of evolutionary dependencies

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 [27]). 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, as this section 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 reason about 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.

## 5 Evolution as dependability

Socio-technical systems [11] 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 [26; 33; 40].

Heterogeneous engineering [31] 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 misunderstood. 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* [24; 25]. Different attributes (i.e. Availability, Reliability, Safety, Confidentiality, Integrity and Maintainability) refine dependability according to complementary properties [25]. The basic impairments of dependability define how *faults* (i.e. the initial cause) cause *errors* (i.e. those parts of system states) that may lead to system *failures* (i.e. deviances of the system service). These identify the chain of mechanisms[5] (i.e. ..., fault, error, failure,...) by which system failures emerge. The *means* (i.e. fault prevention, fault tolerance, fault removal and fault forecasting) for dependability are the methods or techniques that enhance the system ability to deliver the desired service and to place trust in this ability [25].

With respect to dependability the evolution of socio-technical systems transversely affects attributes, means and impairments. On the one hand evolution can enhance system dependability. On the other hand evolution can decrease system dependability. This chapter highlights emergent relationships between evolution and dependability of socio-technical systems. It reviews a taxonomy of evolution, as a conceptual frame-

[5]Note that it is possible to give slightly, but fundamentally, different interpretations to these mechanisms. Different interpretations of the impairments and their mutual relationships highlight that failures emerge differently (e.g. ... error, fault, failure, ...) [15; 26].

work 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.

Dependability models capture evolution in different ways. For instance, fault tolerance models [25; 35] 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 [30] 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. [28; 29]) 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 [40]. These models extend the *Domino* model, which assumes that an accident is the final result of a chain of events in a particular context [19]. Similarly, the *Cheese* model consists of different safety layers having evolving undependability holes. Hence, system failures arise and become catastrophically unrecoverable when they propagate through all the safety layers in place [36]. Despite these models capturing diverse perspectives of the dynamics of system failures, they fail to capture evolution.

Software evolution represents just one aspect of the evolution of socio-technical systems. This work identifies 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 (e.g. inconsistency, limited evolution capturing, system feature emerging, etc.) about system dependability and evolution.

*Example 6.* This example highlights how modelling requirements evolution allows the gathering of evolutionary aspects of socio-technical systems. For instance, the SHEL model [12] 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 allows us to detect these discrepancies. For instance, it is possible to use model checking to

discover mode confusions or automation surprises [37]. 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.

The evolutionary phenomena (e.g. software evolution, requirements evolution, etc.) of socio-technical systems contribute differently 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 1 summarises the different dependability evolutionary perspectives and also proposes some engineering hints.

# 6 Conclusions

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 the 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.

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

**Table 1.** 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 humans 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. |

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 characteri-

sation 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 the evolution of socio-technical systems.

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). Future work aims to acquire practical experience using both the taxonomy and the modelling in industrial settings.

## References


[1] Anderson S, Felici M (2000a). Controlling requirements evolution: An avionics case study. In Koornneef F, 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.

[2] Anderson S, Felici M (2000b). Requirements changes risk/cost analyses: An avionics case study. In Cottam M, Harvey D, Pape R, 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.

[3] Anderson S, Felici M (2001). Requirements evolution: From process to product oriented management. In Bomarius F, 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.

[4] Anderson S, 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.

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

[6] Bergman M, King JL, Lyytinen K (2002a). Large-scale requirements analysis as heterogeneous engineering. Social Thinking - Software Practice, pages 357–386.

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

[8] Bijker WE, Hughes TP, Pinch TJ, editors (1989). The Social Construction of Technology Systems: New Directions in the Sociology and History of Technology. The MIT Press.
[9] Boehm BW, et al. (2000). Software Cost Estimation with COCOMO II. Prentice-Hall.
[10] Chagrov A, Zakharyaschev M (1997). Modal Logic. Number 35 in Oxford Logic Guides. Oxford University Press.
[11] Coakes E, Willis D, Lloyd-Jones R, editors (2000). The New SocioTech: Graffiti on the Long Wall. Computer Supported Cooperative Work. Springer-Verlag.
[12] 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.
[13] Fagin R, Halpern JY, Moses Y, Vardi MY (2003). Reasoning about Knowledge. The MIT Press.
[14] Felici M (2004). Observational Models of Requirements Evolution. PhD thesis, Laboratory for Foundations of Computer Science, School of Informatics, The University of Edinburgh.
[15] Fenton NE, Pfleeger SL (1996). Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press, second edition.
[16] Fitting M, Mendelsohn RL (1998). First-Order Modal Logic. Kluwer Academic Publishers.
[17] Gunter CA, Gunter EL, Jackson M, Zave P (2000). A reference model for requirements and specifications. IEEE Software, pages 37–43.
[18] Halpern JY (2003). Reasoning about Uncertainty. The MIT Press.
[19] Heinrich HW (1950). Industrial accident prevention: a scientific approach. McGraw-Hill, 3rd edition.
[20] Hitchins DK (1992). Putting Systems to Work. John Wiley & Sons.
[21] Hughes AC, Hughes TP, editors (2000). Systems, Experts, and Computers: The Systems Approach in Management and Engineering, World War II and After. The MIT Press.
[22] Hull E, Jackson K, Dick J (2002). Requirements Engineering. Springer-Verlag.
[23] Jarke M (1998). Requirements tracing. Communications of the ACM, 41(12):32–36.
[24] Laprie JC (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.
[25] Laprie JC, et al. (1998). Dependability handbook. Technical Report LAAS Report no 98-346, LIS LAAS-CNRS.
[26] Leveson NG (1995). SAFEWARE: System Safety and Computers. Addison-Wesley.
[27] Leveson NG (2000). Intent specifications: An approach to building human-centered specifications. IEEE Transactions on Software Engineering, 26(1):15–35.
[28] Littlewood B, Popov P, Strigini L (2001). Modelling software design diversity: a review. ACM Computing Surveys, 33(2):177–208.
[29] Littlewood B, Strigini L (2000). Software reliability and dependability: a roadmap. In Finkelstein A, editor, The Future of Software Engineering, pages 177–188. ACM Press, Limerick.
[30] Lyu MR, editor (1996). Handbook of Sofwtare Reliability Engineering. IEEE Computer Society Press.
[31] MacKenzie DA (1990). Inventing Accuracy: A Historical Sociology of Nuclear Missile Guidance. The MIT Press.
[32] MacKenzie DA, Wajcman J, editors (1999). The Social Shaping of Technology. Open University Press, 2nd edition.
[33] Perrow C (1999). Normal Accidents: Living with High-Risk Technologies. Princeton University Press.
[34] Perry DE (1994). Dimensions of software evolution. In Proceedings of the IEEE International Conference on Software Maintenance. IEEE Computer Society Press.

[35] Randell B (2000). Facing up to faults. Computer Journal, 43(2):95–106.
[36] Reason J (1997). Managing the Risks of Organizational Accidents. Ashgate Publishing Limited.
[37] Rushby J (2002). Using model checking to help to discover mode confusions and other automation surprises. Reliability Engineering and System Safety, 75:167–177.
[38] Sommerville I, Sawyer P (1997). Requirements Engineering: A Good Practice Guide. John Wiley & Sons.
[39] Stirling C (2001). Modal and Temporal Properties of Processes. Texts in Computer Science. Springer-Verlag.
[40] Storey N (1996). Safety-Critical Computer Systems. Addison-Wesley.
[41] Weinberg GM (1997). Quality Software Management. Volume 4: Anticipating Change. Dorset House.
[42] Weiss KA, C.Ong E, Leveson NG (2003). Reusable specification components for model-driven development. In Proceedings of the International Conference on System Engineering, INCOSE 2003.
[43] Williams R, Edge D (1996). The social shaping of technology. Research Policy, 25(6):865–899.
[44] Williams R, Slack R, 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.