Chapter 1
An Overview of Requirements Evolution

Neil Ernst, Alexander Borgida, Ivan J. Jureta and John Mylopoulos

Summary. Changing requirements are widely regarded as one of the most significant risks for software systems development. However, in today's business landscape, these changing requirements also represent opportunities to exploit new and evolving business conditions. In consonance with other agile methods, we advocate requirements engineering techniques that embrace and accommodate requirements change. This agile approach to requirements must nonetheless be systematic and incorporate some degree of planning, especially with respect to accommodating quality attributes such as safety and security. This chapter examines the nature of requirements evolution, and the two main problems that it entails. The first is to correctly understand what is changing in the requirements, that is, the elicitation problem. The other is to act on that new information using models and other representations of the requirements, influencing the architecture and implementation of the software system. This chapter first motivates the importance of considering changing requirements in evolving software systems. It then surveys historical and existing approaches to requirements evolution with respect to the elicitation and taking action problems. Finally, the chapter describes a framework for supporting requirements evolution, defining the Requirements Evolution Problem as finding new specifications to satisfy changed requirements and domain assumptions. To motivate this, we discuss a real-life case study of the payment card industry.[1]

[1] Portions of this chapter are adapted from [276].

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_1, © Springer-Verlag Berlin Heidelberg 2014

1.1 Introduction

Most software systems are now expected to run in changing environments. Software developed using an agile methodology often focuses on releasing versions that are only partially completed, at least with respect to the full set of customer requirements. Software must operate in a context where the world is only partially understood and where the implementation is only partially completed. What drives the implementation in this scenario is the requirements, whether represented as user stories, use cases or formal specifications.

Focusing on software evolution only makes sense if one understands what the objectives are for that software. These objectives are themselves frequently changing. Indeed, as we discuss in this chapter, expecting a system's requirements to be constant and unchanging is a recipe for disaster. The 'big design up front' approach is no longer defensible, particularly in a business environment that emphasizes speed and resilience to change [436]. And yet, focusing on implementation issues to the exclusion of system objectives and business needs is equally unsatisfactory.

How does our view of the importance of the system's requirements fit with the historical literature on software evolution? Requirements have long been seen as important, as we shall describe below, in Section 1.2. In 2005, a paper on "challenges in software evolution" [598] highlighted the need to research the "...evolution of higher-level artifacts such as analysis and design models, software architectures, requirement specifications, and so on." More recently, Mens [591] listed several key challenges for software evolution including "How to ensure that the resulting system has the desired quality and functionality?" This question is the motivation for our work in requirements evolution, as we firmly believe that understanding the evolution of non-functional requirements, in particular, will help answer this.

There are three major objections to making requirements more prominent in the study of software evolution. For one, the tangible is easier to study. In many cases, particularly short-term or small-scope projects, requirements are either not used explicitly or stale the moment they are 'finished'. However, this is seldom true of high-value software products, and where it is, typically is symptomatic of a larger software process or organizational pathology. Secondly, in terms of quantity, many change tasks involve low-level corrective maintenance concerns, rather than high-level evolutionary ones. While the numbers of corrective change tasks might be greater, our position is that evolutionary requirements changes are more complex and more costly, and therefore more important, than coping with bug fixes. Finally, requirements and associated changes to the requirements are seen as part of the problem domain and therefore untouchable, much like understanding the organizational objectives might be. We believe that revisiting the problem domain and re-transitioning from problem to solution is of paramount importance in software development.

It is our view that requirements artifacts should drive implementation decisions. In other words, requirements must be tangible, and requirements must be relevant. While they often take the form of work item lists, as is the case in most industrial tools, it is preferable that they be well-structured graphs that represent all aspects of

the requirements problem, capturing stakeholder objectives, domain assumptions, and implementation options. Such models allow for lightweight reasoning (e.g., [277]) where the key challenge is 'requirements repair': re-evaluating available solutions to solve the changed requirements, adding (minimal) new implementations where necessary [627]. We will explain this with reference to the Requirements Evolution Problem, which defines how software should respond to changes in its constituent parts: elements in the specification (the implementation), the system requirements (in the form of goals) and domain knowledge and constraints. We argue this is distinct from the Self-Adaptation Problem (cf. Chapter 7.6), which is concerned with building systems that are self-adaptive, and do not require outside intervention. The Requirements Evolution Problem explicitly supports this guided intervention. This distinction is crucial; while adaptivity is important, at some point systems will need to be managed, such as when they lack implementation to support a particular change—in operating environment, requirements, or capabilities.

In this chapter, we focus on requirements evolution. We begin by introducing the context for considering requirements in the broader field of software evolution. We then turn to the history of research into requirements evolution, including empirical studies. Next, we look at current approaches, focusing first on how industry, and industry tools, have dealt with requirements evolution. We then survey the state of the art in requirements evolution research. To conclude this chapter, we elaborate on one approach to managing changing requirements, with examples drawn from the payment card industry.

The Requirements Problem

As a reference framework, we introduce an ontology for requirements. This work is based on [450] and [449], both of which derive from the fundamental requirements problem of Zave and Jackson [945]. In modern requirements engineering, it is often the case that one distinguishes different kinds of sentences encountered in stating a "requirements problem", according to the "ontology" of the requirements modeling language. In Zave and Jackson's original formulation, the requirements problem is

Definition 1.1. Requirements Problem: Given requirements R (optative statements of desire), a space of possible solution specifications S_P, domain world knowledge W_D, find a subset S of S_P, and software solution finding knowledge W_S such that $W_D, W_S, S \vdash R$.

Domain world knowledge reflects properties of the external world the software operates in, e.g., constraints such as room capacity. Solution finding knowledge reflects how our requirements problem is constructed, so refinement relationships are elements of W_S, that is, the expression "ψ refines ϕ" ($\phi, \psi \in R$) is part of W_S. The Requirements Evolution Problem extends this requirements problem definition to introduce change over one increment of time.

Definition 1.2. Requirements Evolution Problem: Given (i) requirements R, domain knowledge W_D, and (ii) some chosen *existing* specification S_0 (i.e., such that $W_D, W_S, S_0 \vdash R$), as well as (iii) modified requirements problem (δ(R),$\delta(W_D)$,δ(S)) that include modified requirements, domain knowledge and possible tasks, produce a subset of possible specifications $\hat{S}$ to the changed requirements problem (i.e., $\delta(W_D), \hat{S} \vdash \delta(R)$) which satisfy some desired property Π, relating $\hat{S}$ to S_0 and possibly other aspects of the changes.

As an example, consider Figure 1.1. Here we represent a simplified version of a system for managing payments at soccer stadiums (taken from [277]), which must comply with the Payment Card Industry Data Security Standard (PCI-DSS). Payment card issuers, including Visa and Mastercard, developed the PCI-DSS for securing cardholder data. It takes effect whenever a merchant processes or stores cardholder data. We represent this as a set of high-level requirements (ovals) refined by increasingly more concrete requirements, eventually operationalized by tasks (diamond shapes).

In our proposed solution, S_0, the existing approach, consists of tasks "Buy Strongbox", "Use Verifone POS", and "Virtualize server instances", shown in grey shading. In order to be PCI compliant, the requirements evolve to add requirement "Use Secure Hash on Credit Cards" (double-lined oval). This conflicts with our solution (shown as line with crosses), as Verifone terminals do not support this (hypothetically). Instead, we must evolve our implementation to include the task "Use Moneris POS" terminals, i.e., add that to $\hat{S}$ (and retract "Use Verifone POS"), which does not conflict, since it does support secure hashes.

In what follows we use this framework to characterize the challenge of managing evolving requirements. In particular, while software evolution tends to focus on managing largely changes in S, in the field of requirements we are faced with changes in any or all of S, W, R. Furthermore, since these three components are related ($W \cup S \vdash R$), changes in one impact the validity of the inferential relation. For example, changes in requirements R, e.g., from R_0 to R_1, will force a re-evaluation of whether $W \cup S$ still classically entails the satisfaction of R_1.

1.2 Historical Overview of Requirements Evolution

In this section, we survey past treatments of evolving requirements. We begin by exploring how software evolution research dealt with changing requirements. The importance of evolving requirements is directly connected to the wider issue of evolving software systems. While the majority of the literature focused on issues with maintaining and evolving software, a minority tries to understand how changes in requirements impact software maintenance.

The study of software evolution began when IBM researchers Belady and Lehman used their experiences with OS/360 to formulate several theories of software evolution, which they labeled the 'laws' of software evolution. This work was summarized in [510]. These papers characterize the nature of software evolution as an

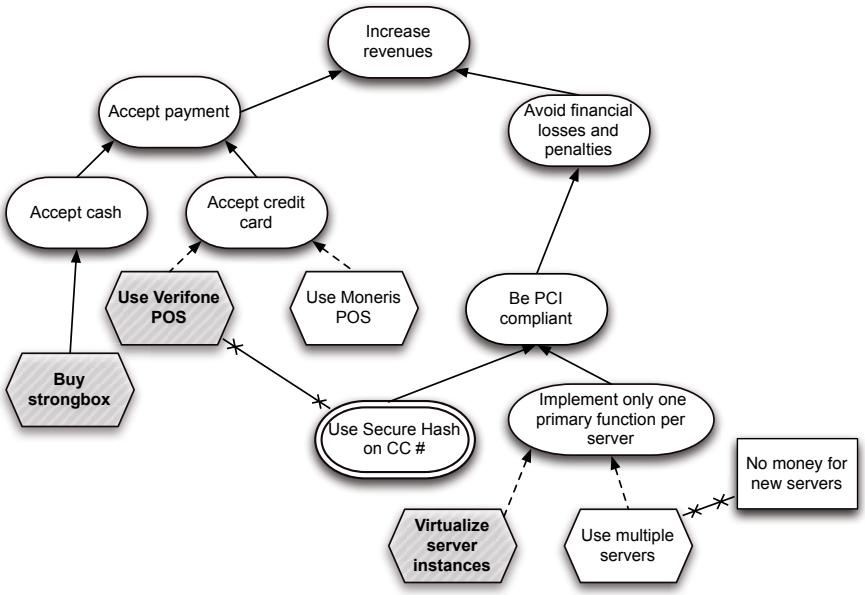

Fig. 1.1: An example of a Requirements Evolution Problem. Shaded grey nodes represent the original solution; the double outlined requirement represents a new requirement. Dashed lines represent alternative refinements, double-crossed arrows conflicts, and regular arrows refinement.

inevitable part of the software lifecycle. 'Inevitability' implies that programs must continually be maintained in order to accommodate discrepancies with their continuously evolving operational environment. One law states that software quality will decline unless regular maintenance activity occurs, and another implies that a system's complexity increases over time. While their work largely focused on implementation artifacts, it clearly acknowledged requirements as a driving force for the corrective action necessary to reconcile actual with anticipated behavior: "Computing requirements may be redefined to serve new uses [91, p. 239]."

Early in the history of software development it became clear that building software systems was nothing like engineering physical artifacts. An obvious difference was that software systems were malleable. Reports suggested a great deal of effort was being spent on maintenance tasks. Basili, writing in 1984, lists 40% [76], and the U.S. National Institute of Standards and Technology report in 2002 claimed industry data show that 70% of errors are introduced during requirements and architecture design, with a rework cost that is 300 or more times the cost of discovering and correcting the errors earlier [797]. Swanson [812] focused on post-release maintenance issues, and looked beyond low-level error fixing (which he termed *corrective* maintenance) to address the issues that Lehman and Belady raised. His work identified "changes in data and processing environments" as a major cause of *adap-*

tive maintenance activity. Swanson's paper marks one of the first times researchers realized that it was not possible to 'get it right the first time'. In some projects, anticipating everything was essential (safety-critical systems, for example); Swanson's insight was that in other projects this was not cost-effective (although it remained desirable).

Development processes still reflected the engineering mindset of the time, with heavy emphasis on up-front analysis and design. US military standards reflected this, since the idea of interchangeable parts was particularly important for military logistics, and the military had experienced enormous software cost overruns. These pressures were eventually realized as the US government's MIL-STD–498, criticized for insisting on a waterfall approach to software development. Afterwards came the slightly more flexible software process standards IEEE/ISO–12207, and IEEE–830, perhaps the best known standard for software requirements to date. But David Parnas's paper on the "Star Wars" missile defence scheme [679] illustrated the problems with this standard's philosophy, many of which come down to an inability to anticipate future requirements and capabilities, e.g. that "the characteristics of weapons and sensors are not yet known and are likely to remain fluid for many years after deployment" [679, p. 1329]. This demonstrated the massive impact unanticipated change can have on software systems, a central concern of this chapter. Indeed, the US military no longer insists that software be developed according to any particular standard [577, p. 42].

In response to the problems with the waterfall approach, iterative models, such as Boehm's 'spiral' model of development [121] called for iterations over system design, so that requirements were assessed at multiple points. However, such process-oriented models can do little to address unanticipated changes if they do not insist on releasing the product to stakeholders. As Fred Brooks notes, "Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. Hence plan to throw one away; you will, anyhow [144]." The point of Brooks's quote is to emphasize how little one can anticipate the real concerns in designing software systems, particularly novel (for its time) systems like OS/360. Instead, development should be iterative and incremental, where iterative means "re-do" (read 'improve') and increment means "add onto", as defined in [498].

1.2.1 From Software Evolution to Requirements Evolution

This section focuses on that part of software evolution that is concerned with changing requirements or assumptions (i.e., the components of the requirements problem which are in R or W). Historically, some researchers have turned to focus in detail on this relationship between requirements and evolution of software. Not all maintenance activities can be said to result in 'software evolution': for instance, when designers are correcting a fault in the implementation (S) to bring it (back) into line with the original requirements (which Swanson called 'corrective maintenance').

Chapin [169, p. 17] concludes that evolution only occurs when maintenance impacts business rules or changes properties visible to the customer.

Harker et al. [363] extended Swanson's work to focus on change drivers with respect to system requirements (summarized in Table 1.1), because "changing requirements, rather than stable ones, are the norm in systems development [363, p. 266]." He characterized changes according to their origins. At this point, requirements engineering as a distinct research discipline was but a few years old, and an understanding was emerging that the importance of requirements permeated the entire development process, rather than being a strictly 'up-front' endeavour.

Table 1.1: Types of requirements change [363]

Type of requirement		*Origins*
Stable	Enduring	Technical core of 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

As an aside, it is interesting to ponder whether there is in fact such a thing as an enduring requirement, as defined by Harker et al. A useful analogy can be derived from Stuart Brand's book on architectural change in buildings [138]. He introduces the notion of shearing layers for buildings, which distinguish change frequency. For example, the base layer is *Site*, which changes very little (absent major disasters); *Skin* describes the building facade, which changes every few decades, and at the fastest layer, *Stuff*, the contents of a building, which changes every few days or weeks. The implication for requirements is that good design ought to identify which requirements are more change-prone than others, and structure a solution based on that assumption. There probably are enduring requirements, but only in the sense that changing them fundamentally alters the nature of the system. For example, if we have the requirement for a credit card processing software to connect to the customer's bank, such a requirement is sufficiently abstract as to defy most changes. On the other hand, we can easily foresee a requirement "Connect to other bank using SSL" changing, such as when someone manages to break the security model. We posit that the enduring/changing distinction originates in the abstractness of the requirement, rather than any particular characteristic.

The above taxonomy was expanded by the EVE project [487]. Lam and Loomes emphasized that requirements evolution is inevitable and must be managed by paying attention to three areas: monitoring the operating environment; analysing the impact of the system on stakeholders, or on itself; and conducting risk management exercises. They proposed a process model for systematizing this analysis.

Changes to requirements have long been identified as a concern for software development, as in Basili [76]. Somerville and Sawyer's requirements textbook [787]

explicitly mentions 'volatile' requirements as a risk, and cautions that processes should define a way to deal with them. Their categorization closely follows that of Harker et al.

Several historical research projects in the area of information systems modeling have touched on evolution. CIM [149] labeled model instances with the time period during which the information was valid. Furthermore, CIM "should make incremental introduction and integration of new requirements easy and natural in the sense that new requirements should require as few changes in an existing model as possible [149, p.401]." Little guidance was given on how to do this, however. In a similar vein, RML [350], ERAE [264] and Telos [628] gave validity intervals for model instances using logic augmented with time arguments. These modeling languages were oriented to a one-off requirements model that can then be used to design the system (rather than allowing on-the-fly updates and inconsistencies during run-time). In other words, these methodologies assume complete knowledge of the system, e.g., the precise periods for which a concept is applicable.

Research has also considered the issue of maintaining consistency in requirements models. Models can be inconsistent when different users define different models, as in viewpoints research. The importance of permitting inconsistency in order to derive a more useful requirements model was first characterized by Easterbrook and Nuseibeh [269]. We return to the use of formal logic for managing evolving requirements in Section 1.4. Zowghi and Gervasi explain that "Increasing the completeness of a requirements specification can decrease its consistency and hence affect the correctness of the final product. Conversely, improving the consistency of the requirements can reduce the completeness, thereby again diminishing correctness [955]." With respect to changes in R, then, there seems to be a tradeoff between making R as detailed as possible and making R as consistent as possible. In early requirements analysis where we suspect W will change (for example, in mobile applications) incompleteness should be acceptable if it supports flexibility - we would rather have high-level consistency with incomplete requirements.

Finally, one could consider the elaboration (i.e., increasing the completeness) of the initial requirements model, from high-level objectives to lower-level technical requirements, as 'evolving' requirements (as in [35]); we focus on requirements models for which the elicitation necessary for a first release is assumed to be completed, and then changes, rather than the process of requirements elicitation at an intermediate point in time.

1.2.2 Empirical Studies of Requirements Evolution

The focus of this section is on research projects which conducted empirical studies using industrial Requirements Evolution Problems. Many industrial case studies focus on source code evolution, and little attention is paid to the requirements themselves (which presumably are driving many of the changes to source code). This is typically because requirements are often not available explicitly, unlike source

code. This is particularly true in the case of open-source software. Nonetheless, the following studies do show that, when available, the problem of requirements change is important and not particularly well understood.

The SCR (Software Cost Reduction) project at the U.S. Naval Research Laboratory was based on a project to effectively deliver software requirements for a fighter jet, the A–7E. In a retrospective report on the project [23], which includes updates since the initial release in 1972, Chapter 9 of the report lists some anticipated changes. Of interest is that these changes, while anticipated, are not very detailed, and some invariants are assumed (which we would term domain assumptions, such as "weapon delivery cannot be accurate if location data is not accurate"). This early work identified the criticality of understanding how R could and did change, and that such changes needed to be anticipated.

Chung et al. [187] looked at the problem of (non-functional) requirements evolution at Barclays, a large bank. After introducing a modeling notation, they considered the change of adding more detailed reporting on accounts to customers. The paper showed how this change can be managed using their modeling notation, leading to new system designs. In our parlance, they considered changes in all of R, W, S. In particular, the paper focused on tracking the impact of a change on other non-functional properties of the system, such as accuracy and timeliness (i.e., quality attribute requirements in R). Their notation allows analysts to model priorities and perform semi-automated analysis of the updated model. With respect to our property Π, this study used the degree to which a solution $\hat{S}$ satisfied quality attributes as the property over which to evaluate solution optimality. The paper concludes with some principles for accommodating change.

In [36], Anton and Potts looked at the evolution of the features offered by a customer-centric telephony provider. The paper traced, using historical documents, which features were available and how this changed over time. In particular, the paper focused on the consequences of introducing certain features, with the objective of reducing the effort of providing a new service to customers. This survey was end-user oriented as it focused on how features appeared to users of telephone services, not other businesses or the internal feature requirements. Changes in W are related to subsequent changes in S (features are properly parts of the solution), but there is little or no role for explicit members of R, except as reverse-engineered. One can reverse engineer changes in R by inference: if S changes as new features are added, and the authors show it was in response to some initiating change from the domain knowledge W (such as customer usage), then we can infer a change in R (since $W \cup S \vdash R$).

The Congruence Evaluation System experience report of Nanda and Madhavji [632], while not conducted on industrial data, did shed some useful light on the Requirements Evolution Problem. This was an academic-built proof of concept that ultimately failed. In their analysis, Nanda and Madhavji explicitly note that changes in W, which they term "environmental evolution" was a major factor. They particularly note how difficult it was to communicate these changes into direct impacts on the requirements, as they were typically not monitored explicitly.

Anderson and colleagues conducted a series of case studies of changing requirements [31, 32], focusing explicitly on changes in R. Their experiences led to the development of a taxonomy for requirements evolution. The case studies focused on smart cards and air traffic control, and spurred the development of the Requirements Maturity Index, in order to measure how frequently a particular requirement changed. However, the index did not provide for different measures of requirements value or importance, i.e., there was no explicit notion of comparison with respect to Π.

Tun et al. [860] used automated extraction techniques to obtain problem structures from a large-scale open source project (the Vim text editor). They concede that requirements may not be easily extracted, but contend that these problem structures shed some useful light on how one might implement new requirements. These problem structures are essentially triples of W, S, R, with particular focus on looking at S in order to attempt to derive the other two. The challenge with all studies of purely source code (i.e., S) is that one must to some extent speculate about whether changes are coming from the domain knowledge W or from the requirements changing.

There is relevant work in the Mining Software Repositories community on extraction of requirements from project corpora, most recently the work of Hindle et al. at Microsoft [404]. They correlated project specifications to source code commits and were able to identify related changes. In this context, Hindle et al. used the project specification as the representation of R and the code commits as insight into the implementation S. The chief problem with open-source project is that requirements are rarely made explicit. Instead, they occur as user stories or prototyped features. In [279] we looked at techniques for extracting a set R from these issue trackers. See also Chapter 5 later in this book, on repository mining.

Many studies of changing requirements have focused on software product lines. We do not discuss them here, since Chapter 9 goes into them extensively. Herrmann et al [392] used an industrial case study to identify the need for "delta requirements", requirements which must be added subsequent to software delivery, and then took an existing methodology and extended it to incorporate modeling techniques for delta requirements. The advantage of defining delta requirements is that it permits baselining one's core functionality (similarly to product lines), and then extending these as necessary.

Ideally, of course, one would minimize changes before they occur, rather than needing to manage changes afterwards. The issue of changing requirements in the highly formal requirements environment of spacecraft design was considered in [636], with the aim of minimizing the number of downstream changes required, as each change is potentially very costly to re-implement. The authors proposed a technique, semantic decoupling, for modeling requirements R to minimize the dependencies between abstraction levels. In semantic decoupling, one tries to minimize the dependencies between related software modules (S), so that changes in R will not be as wide-ranging. This of course requires a reasonably complete definition of a traceability matrix in order to identify the relationships (which does typically exist in the domains they study).

This sampling of academic case studies of changing requirements in industrial settings has provided some clear examples of the importance of requirements evolution. In domains as varied as spacecraft, smart cards, and phone features, changing requirements are clearly a major concern for business, and the source of much cost and effort.

1.3 A Survey of Industry Approaches

It is useful to consider the treatment of changing requirements in industry settings, as a way to understand the current practices, and how these might inform research proposals. Industrial tools have a strong focus on interoperability with office software like Microsoft Word, because a common use-case for these tools is generating documentation. Furthermore, these tools are not the whole story, as many industry consultants (e.g., [504, 922]) focus as much on managing change through methodology as through tools. This means creating a change process which might incorporate reviews, change tracking, prioritization meetings, and so on.

1.3.1 Standards and Industry

IEEE Standard 830 [421], which describes a standard for "Software Requirements Specification" (SRS), is the culmination of the strict specification approach, what some have derisively called "Big Requirements Up Front". It lays out in great detail the standard way for describing "what" must be built. Section 4.5 of the standard addresses evolution, which it recommends managing using notation (marking requirements as "incomplete") and processes for updating the requirements. As with most standards, this is reasonable in mature organizations, but prone to problems if these key ideas are not followed. Furthermore, completeness and stability are often orthogonal concerns. The standard acknowledges that evolutionary revisions may be inevitable.

1.3.2 Requirements Management Tools

Commercial tools have generally done a poor job supporting change. IBM DOORS[2] and IBM Requisite Pro are document-centric tools whose main interface consists of hierarchical lists of requirements (e.g., "R4.2.4 the system shall ..."). Traceability is a big feature of such tools, and requirements can be linked (to one another and to other artifacts, such as UML diagrams). Multiple users are supported, and changes

[2] http://www-01.ibm.com/software/awdtools/doors/

prompt notification that the requirement has changed. Version control is important: each requirement is an object, and the history of that object is stored, e.g., "modified attribute text" on DATE by USER. In DOORS, one can create requirements baselines which are similar to feature models. One can extend the baseline to create new products or changes to existing projects. It is not clear what the methodology for defining a baseline is.

The tool focus of Blueprint Requirements Center[3] is agile, with strong support for simulation and prototyping. Workbenching requirements scenarios is important in Blueprint. Workbenching or simulation helps analysts understand all the potential variations, as well as giving something concrete to the business user before costly implementation. Blueprint also focuses on short-cycle development, allowing requirements to be broken into sprint-specific stories or features. What both Blueprint and the IBM suite miss, however, is a way to combine requirements management with workbenching integrated into a framework for evaluating change impacts.

1.3.3 Task Managers

An increasingly popular strategy in industry is to forego IEEE specification conformance in favour of lightweight task management tools. This might be described as the agile approach to requirements: treating requirements as tasks that must be carried out. Jira, from Atlassian Software[4], is a commonly-used tool in large-scale projects. Jira allows one to manage what is essentially a complex to-do list, including effort estimation, assignment, and some type of workflow management (e.g., open issue, assign issue, close issue). Similar tools include Bugzilla, Trac, and IBM's Rational Team Concert. More recently, Kanban [30] has made popular visual work-in-progress displays, the most basic of which are whiteboards with life-cycle phases as swimlanes. These tools are well-suited to the deliberate reduction of documentation and adaptive product management that agile methodologies such as Scrum or XP recommend. Leffingwell [503] gives a more structured approach to agile requirements engineering, managing changes using time-boxed iterations (e.g., 2 week cycles) at the boundaries of which the team re-prioritizes the user stories to work on for the next cycle. In this fashion, changes in the domain knowledge W and new requirements R can be accommodated on a shorter time-frame than a model with change requests. This constant iteration only works well with a robust set of tests to verify the requirements were correctly implemented, e.g., using unit and system tests, but as important is some automated acceptance tests using, e.g., Behavior-Driven Development (BDD).

[3] http://www.blueprintsys.com/products/

[4] http://www.atlassian.com/software/jira/

1.3.4 Summary

Particularly for smaller organizations, requirements are not treated at a high level, often existing as an Excel spreadsheet or maintained implicitly [46]. Furthermore, the transition to agile software development has made one of its chief priorities the reduction of unnecessary documentation ("working software over comprehensive documentation"[5]). It is an open and important research question whether omitting at least some form of explicit requirements model is sustainable in the long-term.

The tools we have described work well for managing low-level tasks, such as fixing specific bugs. However, connecting the design and roadmapping component of product management with the specifics of task management is more difficult. While some might use tools like Confluence or other wikis for this knowledge-management task, spreadsheets are still very popular for tracking lists of possible features. What is missing is a higher-level view of "why" changes are being made, and what impact those changes might have on satisfying the requirements. A tool which can preserve the overall requirements model throughout the lifecycle is necessary. That is not to say such an approach could not be integrated into a tool like IBM DOORS. Indeed, there is a lot of work on integrating requirements tools, task managers, code repositories and so on using product lifecycle management (PLM) or application lifecycle management (ALM). The emerging standard for Open Services for Collaboration (OSLC)[6] is one initiative that looks to overcome the traditional silos.

1.4 Recent Research

We now survey some of the latest research in requirements evolution. In many cases, research has focused most on eliciting requirements and potential changes, and less on how such models/representations would be used to adapt software post-implementation. Interest in the notion of requirements at run-time has greatly increased recently, however, and we touch on this below. There are overlaps with work on adaptive software (see Chapter 7.6 later in this book) and model-driven evolution (Chapter 2). To conclude, we introduce two summary tables showing how the individual work addresses elements of the Requirements Problem, as well as an explanation of where gaps exist between research and practice.

[5] http://agilemanifesto.org/

[6] http://open-services.net/

1.4.1 Problem Frames Approach

We mentioned the empirical study of Tun et al. [860] earlier. This work builds on the seminal problem frames approach of Michael Jackson [431] to extract problem frames from existing software systems in order to recover the original requirements. A problem frame captures a particular set of elements involved in the Zave and Jackson approach to the requirements problem: $W, S \vdash R$. For example, the text editor Vim has a feature "Spell Completion". From the requirement description, Tun et al. reconstruct the problem diagram using problem frame syntax: the requirement is on the right, "complete word automatically", linked with shared phenomena including "keyboard user" and "buffer", and finally, to the machine element implementing "Spell Completion" (the feature). Matching related problem diagrams can show feature interaction problems, in this case, where two features both use the shared phenomena of "buffer". These feature interactions are difficult to manage and can be a large source of problems.

Another project by Tun et al.[859] uses problem frames to identify common problematic design patterns, and to then transform that feature using a catalog. The idea is to support evolution of features using well-known patterns to avoid feature interaction problems. For example, if I know that my buffer is shared by two features, I can apply a pattern like "Blackboard" to solve the potential problems. Similarly, Yu et al. [942] use problem frames in the context of security requirements. Their tool, OpenArgue, supports argumentation about requirements satisfaction that can be updated as more information arrives. As with the requirements evolution problem we defined, this approach seeks to reason about what the implications of this new information are.

1.4.2 Extensions of the NFR Framework

The NFR model, introduced in [186], represented a qualitative approach to modeling system requirements as refinements of high level objectives, called goals. This has been extended to reason about partial goal satisfaction in a number of ways. To begin, Giorgini et al. [327] and Sebastiani et al. [760] formalized a variant of the NFR framework's qualitative approach, the idea being that qualitative reasoning is better suited to up-front problem exploration. Their tools (e.g., GR-Tool[7]) can reason over qualitative models and generate satisfying alternatives. One can leverage this approach to incrementally explore evolving requirements problems.

What was not well understood was how to turn these into specifications. From the evolution point of view, work on alternatives and variability in goal modeling (e.g., [520], [497]) allows these qualitative models to capture context-driven variability, a point also made in Ali et al. [18], who make the case that requirements variability necessitates the monitoring of the operating contexts. This monitoring

[7] http://troposproject.org/tools/grtool/

information is then used to inform a designer about possible revision options. In all these cases the main contribution to requirements evolution is in eliciting alternative solutions and extending the system specification with annotations for monitoring for violations of these models. Dalpiaz et al. [215] also introduced qualitative requirements variability, but in the area of dynamic reconfiguration. This proposal goes from modeling and elicitation to system specification *over time*, i.e., not just for the initial design but also once the system has been released.

1.4.3 Run-time Adaptive Requirements

Work in the area of adaptive requirements focuses on understanding how to build requirements-based systems that are responsive at run-time to system changes. In particular, the notion of "requirements at runtime", explored in a series of workshops at the Requirements Engineering conference (`requirements-engineering.org`), introduced the notion of using requirements models to drive system changes. See also the chapter on adaptive software later in this book (Chapter 7.6). One thing that is necessary for run-time evolution is the ability to understand what is changing. Qureshi et al. [705] define a set of ontological concepts for managing run-time adaptation to the changes in the requirements problem. The main achievement is the addition of context to requirements problems, in order to suggest variations when contexts change. Another approach is to loosen the formal representation: In the RELAX framework [920], a language is designed to specifically manage "the explicit expression of environmental uncertainty in requirements". When something changes in W (the world), for example, a new device appears on a mobile ad-hoc network, the RELAX language can define service levels which satisfy higher level requirements (e.g., "connect to AS MANY devices as possible" as opposed to "connect to ALL devices"). In similar fashion, Epifani et al. [275] use a formal model and a Bayesian estimator to adapt numeric parameters in the specification at run-time. This allows them to set an initial model of the system and then fine-tune various parameters as more information is collected. This is a little like learning the true system requirements, rather than specifying them all at once.

1.4.4 KAOS-based Approaches

A major contribution to the RE literature is the KAOS goal modeling framework, first introduced in [217]. The original focus was on a methodology and tool for goal-based decomposition of requirements problems. The original work has been extended in a number of ways. One direction has considered the importance of alternatives in system design. From an evolution perspective, variability and alternatives support resiliency in two ways when change is encountered. First, the upfront analysis supports enumeration of possible scenarios that need to be handled (for

example, the obstacles encountered in [872]). Second, variants can be managed as a form of product line, and called upon if and when things change (see Chapter 9 for more on product lines and requirements). Later work [515] introduced probabilistic techniques for monitoring the partial satisfaction of goals in KAOS models. As designers explore the solution space, numeric measures are used to evaluate the value of a given configuration. Not covered in detail in the paper is how this model would be adjusted at run-time and used to evolve the specification *S*, but some of the KAOS concepts, and in particular its formalism, have found their way into problem frames work. In [871], van Lamsweerde discusses how one might compare alternative models of requirements and systems to be designed therefrom.

1.4.5 Paraconsistent and Default Logics

Several requirements modeling approaches rely on formal logic explicitly (KAOS also uses a formal temporal logic, but it is not the focus of the KAOS-based approaches described above). Here we review two approaches.

Default logic approaches, appearing in [956] and [325], rely on David Poole's Theorist system [691] to define what *must* be (typically the World knowledge) and what *might* change, represented by initial defaults. The connection to the requirements model is two-fold: the selection of the order in which requirements are considered for revision, and the ability to 'downgrade' requirements to default (preferred) status rather than 'mandatory' status. Default logic is non-monotonic in that asserted (TOLD) facts can later be contradicted and no longer concluded; for example, the sentence "requirement R is refined by task T" can be over-ruled if new information is discovered that says, for example, that "requirement R has no refinements". In classical logic, as long as the original sentence remains in the theory, it can be deduced.

Closely aligned with this perspective is the REFORM framework of Ghose [325], which identifies three main properties for a system managing evolution:

1. distinguish between what are called *essential* and *tentative* requirements;
2. make explicit the rationale for satisfying a requirement (refinements);
3. make explicit the tradeoffs for discarding a requirement when the requirements model changes.

Ghose [325] also defines some useful principles for handling changes to the requirements:

1. make *minimal* changes to the solution when the problem changes;
2. make it possible to ignore the change if the change would be more costly than ignoring it;
3. support *deferred commitment* so that choosing a solution is not premature.
4. maintain discarded requirements to support requirements re-use.

They go on to implement these ideas in a proof-of concept system for managing requirements. One issue to consider in such non-monotonic systems for requirements is that reasoning from events to explanations is abductive, and therefore in the NP-hard class of problems. Abductive reasoning is to reason 'backward', using observations and a background theory to derive explanations, as opposed to deductive reasoning, which uses a background theory and an explanation to derive possible observations.

Another approach to managing change is *to support paraconsistent reasoning*, that is, reasoning in the presence of contradictory evidence without trivially concluding everything, as in classical logic. This is vital in handling evolving requirements since one common occurrence is that a fact previously asserted as true is then found to be false. For example, stakeholders might indicate initially that requirement R_x must be satisfied, but at a later time, perhaps the stakeholders realized they did not need the requirement. In a formal model we would have $\{R_x, \neg R_x\}$, a classical inconsistency.

In the RE domain, tolerating inconsistency is essential, for reasons listed by Nuseibeh et al. [648]:

1. to facilitate distributed collaborative working;
2. to prevent premature commitment to design decisions;
3. to ensure that all stakeholder views are taken into account;
4. to focus attention on problem areas [of the specification] .

Hunter and Nuseibeh [413] use Quasi-Classical Logic (QCL), an approach to reasoning in the presence of inconsistency which labels the formulas involved. This also permits one to identify the sources of the inconsistency and then, using their principle of "inconsistency implies action", choose to act on that information, by, for example, removing the last asserted fact, perhaps using principles such as Ghose's, above. An example from the London Ambulance case has a scenario where, based on the information gathered, one can derive both "dispatch Ambulance 1" and "do not dispatch Ambulance 1". Two useful capabilities emerge from labeled QCL: one can continue to derive inferences not related to this inconsistency, for example, that Ambulance 2 needs a safety check; and secondly, to understand the chain of reasoning that led to the inconsistency, and resolve it according to meta-level rules.

Our work [278] used paraconsistent reasoning to manage evolving requirements problems. We defined a special consequence relation $|\sim$ which used a form of maximally consistent subset reasoning to draw credulous conclusions about whether a given high-level requirement could be satisfied, or alternately, which specification to implement in order to satisfy those requirements.

Paraconsistent reasoning (whether using defaults, QCL, or other approaches such as multi-valued logics) supports evolving requirements by mitigating the challenge of conflicting and possibly inconsistent specifications, whether in the World, Requirements, or Specification. While there is a computational complexity concern, practically speaking this is less of an issue as processing speeds have increased.

1.4.6 Traceability Approaches

Traceability refers to the linkages between (in our case) requirements and downstream artifacts such as code, models and tests. The importance of traceability with respect to software evolution is to support the identification and impact of the changes to the requirements. A number of current approaches use traceability to manage requirements evolution.

In the work of Charrada and Glinz [171], outdated requirements are identified based on changes in the source which are believed to impact requirements. Such changes might, for example, include the addition of a new class or method body, which is likely a new feature being added. This addresses the issue of requirements drifting from the actual state of the source code. This approach relies on machine learning techniques to identify statistically likely candidates, which in general falls into the area of mining software repositories. The basic notion is to gather a large body of data, including source code, requirements documents (if any), tests, emails, etc. Machine learning techniques such as Latent Dirichlet Allocation (see Chapter 5) can then be used to extract interesting labels for that data, including which quality requirements are affected, as in Hindle's work [404]. There is some promise in these approaches, but the major stumbling block is to gather useful data in large enough volumes (typically in the millions of records) that the statistical techniques will be sufficiently accurate. As one might imagine, identifying requirements in source code is tricky without a good set of requirements documents to go from.

Should sufficient data not be available, one is forced to leverage human knowledge to a greater extent. Herrmann et al. [392] use the information about the previous incarnation of the system to derive "delta requirements", which specify only those changes to the requirements necessary to implement the system (we might think of this as the set represented by $\delta R \setminus R$). The challenge with this approach is to correctly characterize the existing system's initial requirements.

Welsh and Sawyer [913] use traceability to identify changes that affect dynamically adaptive systems (DAS). They include five possible changes to a DAS that might need to be accommodated:

- environmental change (a change to W_D)
- broken assumption (an incorrect fact in W_D)
- new technology (new elements in S)
- consequential change (changes to the inferences drawn from $W \cup S$)
- user requirements change (changes to R)

Traceability techniques should somehow identify which type of change is occurring and what implications that change has for the other elements of the system. Welsh and Sawyer extend the i* strategic rationale framework [938] to annotate the models with possible changes and impacts. The primary contribution is to support elicitation and modelling.

1.4.7 Feature Models

Feature models are covered in greater detail in Chapter 9. Techniques for dealing with changes to feature models, including the product lines which are typically derived from the feature models, overlap with the management of requirements evolution. Requirements researchers typically consider feature models to focus on the user-oriented aspects of a system, i.e., be designed with marketable units in mind. Requirements as we define them, however, would also consider non-functional properties of the system (which are not necessarily user-oriented, such as maintainability) and features which may not be relevant to product lines.

That being said, the techniques for managing evolution in feature models are relevant to requirements evolution as well.

1.4.8 Summary

Table 1.2 is a summary of the focus of the approaches discussed, based on whether the approach emphasizes changes in domain knowledge W_D, specification/implementation S, requirements R, or some property Π that can be used to compare approaches. The most glaring omission are techniques for quantifying the difference between various solutions (that is, defining properties Π), although this work has been the subject of work in search-based software engineering (Chapter 4). Applying optimization techniques like genetic algorithms to the Requirements Evolution Problem seems a fruitful area of research.

We said at the beginning of this chapter that managing evolving requirements could be broken down into elicitation and modeling and turning those representations into software. Most of the approaches we discussed focus on the modeling and analysis aspects of the problem. There is unfortunately little work on taking the frameworks and applying them to industrial requirements problems. Part of the challenge is that a lot of industries simply do not manage requirements in a manner which would permit, for example, delta requirements to be generated. Another is that academic tools for the most part ignore the vastly different scale of industrial challenges (Daimler, for example, has DOORS models with hundreds of thousands of objects).

An emerging trend in requirements evolution is the linkage to dynamic, self-adaptive systems (cf. Chapter 7.6). Researchers are increasingly looking beyond the traditional staged lifecycle model where requirements are used to derive a specification and then ignored. Instead, requirements, and other models, are believed to be useful beyond the initial release of the software. Most of the research to date has identified challenges and future work that must be dealt with before we can realize the vision of "requirements at run-time". For example, Welsh and Sawyer [913], Ghose [325], and several others focus on understanding the nature of the problem using classification techniques.

Table 1.2: Research approaches compared with respect to the Requirements Evolution Problem components. (●: covered; ○: partial coverage; –: not covered, na: tool mentioned but not available.)

Approach	Paper	Domain changes δW	Specification changes δS	Requirements changes δR	Solution comparison Π	Tool support	Empirical evidence
Problem Frames	Tun et al. [859]	●	●	●	–	–	–
	Tun et al. [860]	○	●	●	–	[a]	●
	Yu et al. [942]	○	●	●	–	[b]	○
NFR extensions	Sebastiani et al. [760]	●	○	●	●	[c]	○
	Lapouchnian et al. [497]	●	–	●	●	–	○
	Ali et al. [18]	●	–	●	●	–	–
	Dalpiaz et al. [215]	●	–	●	●	–	●
Adaptive Requirements	Qureshi et al. [705]	○	○	●	○	–	–
	Whittle et al. [920]	●	●	●	●	–	●
	Epifani et al. [275]	●	●	●	●	–	●
KAOS-based	van Lamsweerde and Letier [872]	●	–	●	○	[d]	●
	Letier and van Lamsweerde [515]	●	○	●	●	[d]	●
	van Lamsweerde [871]	○	–	●	●	[d]	○
Paraconsistent	Zowghi and Offen [956]	○	–	●	●	na	–
	Ghose [325]	○	–	●	○	–	–
	Hunter and Nuseibeh [413]	●	○	○	–	na	○
	Ernst et al. [278]	●	–	●	○	[e]	●
Traceabiity	Charrada and Glinz [171]	○	●	●	○	–	○
	Hindle et al. [404]	○	●	●	–	–	●
	Herrmann et al. [392]	○	○	●	●	-	●
	Welsh and Sawyer [913]	●	○	●	●	[f]	●

[a] http://mcs.open.ac.uk/yy66/vim-analysis.html

[b] http://sead1.open.ac.uk/pf

[c] http://troposproject.org/tools/grtool/

[d] http://www.objectiver.com/index.php?id=25

[e] http://github.com/neilernst/Techne-TMS

[f] http://is.gd/7wP7Sj

One of the seminal papers in characterizing evolutionary systems is that of Berry et al. [100]. In it, the authors argue that for dynamically adaptive systems requirements engineering is continuous. Systems must understand what objectives are paramount at a given point in time, what constraints exist, and how to act to achieve their objectives. They therefore argue for four levels of adaptivity:

Level 1 Humans model the domain, W and possible inputs to a system S.

Level 2 The given system S monitors W for its inputs and then determines the response. This is the domain of self-adaptive software research, such as Epifani et al. [275] or Whittle et al. [920].

Level 3 Humans identify adaptation elements for the set of systems. This is what variability modeling for goal models does, for example Dalpiaz et al. [215].

Level 4 Humans elicit mechanisms for self-adaptation in general.

Berry et al.'s classification allows us to understand the general trajectory for research into requirements evolution. It moves beyond level 1, which is interested in inputs and outputs for a specific system, increasingly focusing instead on adapting and evolving the software *in situ*, based on a set of observations and a formalism for responding to the inputs. Unknown unknowns, the inputs not modeled for reasons of either cost or ignorance, will still bring us back to level 1.

While this is encouraging in terms of software that will be more resilient, one question that is commonly left unanswered in research is the issue of responsiveness. Particularly in formal analysis, we can quickly run up against fundamental complexity results that render complete optimal solutions infeasible, since exponential algorithms seem to be the only alternative. While the performance of such algorithms has improved with advances in inference engines, processing speed and parallelization, it is very much an open question as to how much analysis is possible, particularly in the 'online' scenario. Hence, a number of researchers focus on incremental or partial approaches to analysis. It is important to keep in mind how well the proposed evolved design will work under realistic scenarios.

Table 1.3 presents a matrix of where research is needed in requirements evolution. These are common RE research themes (see Nuseibeh and Easterbrook [647], Cheng and Atlee [178] or van Lamsweerde [870] for an overview of requirements engineering research in general) but in this case specifically to do with evolving requirements. We rank the challenges according to industry adoption and interest, existing research interest, and finally, challenges and obstacles to further research. One particularly important area, emerging from industry, is a general need for more applied and empirical studies of requirements evolution, in particular in agile projects using lightweight tools. Adaptivity is another emerging research area; in this table, the research area most under-served is understanding how Requirements Evolution Problems can be elicited and analyzed, i.e., what future capabilities will a system require, and how to monitor and understand the possible changes that might occur.

Table 1.3: Research opportunities in requirements evolution (**H** - High, **M** - Moderate, **L** - Low)

Research area (*Evolution and...*)	**Industry adoption**	**Research interest**	**Challenges**
Elicitation	**M** numerous approaches	**L** Most approaches focus on stable systems.	Longitudinal case studies required.
Analysis	**L** conducted by business personnel	**L** most work focuses on adaptation - what to do next.	Understanding system context.
Modeling	**L** models mostly informal	**M** numerous studies of formalization of change, e.g. [277]	Scalability; ease of use by industry.
Management	**M** most tools support some form of impact analysis, but at a simple level.	**H** Numerous frameworks and techniques.	Study mainly on greenfield systems. Little empirical validation.
Traceability	**H** widely seen as important.	**H** see traceability workshops e.g. [693]	Trace recovery; scalability
Empirical research	n/a	**M** - increasing amount of empirical validation in research papers	Reality is messy; Industry reluctance to share data

1.5 A Framework for Requirements Evolution

This section considers strategies for managing Requirements Evolution Problems (REP) at the next stage in the software lifecycle: implementation. In our work, we represent requirements as goals G according to goal-oriented requirements engineering [870]. Recall the definition of the Requirements Evolution Problem needed to relate changes in requirements R, domain knowledge W_D, and solutions S to the existing solution (or find such a solution, if there isn't one). In any event we supported solution comparison by the use of a desired property Π, relating $\hat{S}$ to S_0 and possibly other aspects of the changes. Π, in other words, allows us to define a partial order over potential $\hat{S}$.

We will store instances of these elements (i.e., a specific goal instance such as "system will allow user registration") in a knowledge base called REKB. The REKB can answer questions about the items stored in it. In [277] we discuss this in more detail, but the essential operations (ASK questions in knowledge base parlance) include:

ARE-GOALS-ACHIEVED-FROM Answers True if a given set of tasks in REKB can satisfy a given set of goals. This is the "forward reasoning" problem of Giorgini et al. [327].

MINIMAL-GOAL-ACHIEVEMENT REKB discovers Σ, the set of sets S of tasks which minimally satisfy desired goals. This is the (abductive) backward reasoning problem of Sebastiani et al. [760].

GET-MIN-CHANGE-TASK-ENTAILING This operation produces solutions to the Requirements Evolution Problem. Given an initial solution S and a set of desired goals, find a set of minimal sets of tasks which are not dominated for some criterion of minimality and satisfy the new requirements problem. The key issue is to define what the minimality criterion might be for a new solution, which we discuss below.

We concentrate on those changes which are unanticipated. By 'unanticipated' we mean that there exists no mechanism in the implementation specification S to accommodate these changes. This clearly positions the Requirements Evolution Problem as quite distinct from the Self Adaptation Problem (SAP). With respect to the aspects of G, W, and S, the SAP is to accommodate changes in W,G by creating a suitably adaptive $\hat{S}$ *ab initio*. In other words, unlike the Requirements Evolution Problem, self-adaptation approaches do not modify requirements themselves, but rather choose predefined tasks in S to accommodate the changes. This is what RELAX [920] is doing: using a fuzzy logic to define criteria for satisfying requirements that may be accomplished by different tasks/monitors.

It is also becoming clear that with a suitably flexible framework and a wide pool of services, an adaptive specification can be used to select services that satisfy our changed goals or domain assumptions. Since these services can be quite heterogeneous, there is a continuum between *adapting* and *evolving*. The essential distinction is the extent to which the changes are anticipated in the implementation.

There are two key concerns in the Requirements Evolution Problem:

1. What do we do when new information contradicts earlier information? This is the problem of **requirements problem revision**.
2. What solutions (sets of tasks) should we pick when the REKB has changed and been revised? This is the problem of **minimal solution selection**.

We discuss these below, after introducing our motivating case study.

1.5.1 The Payment Card Industry Example

As we discussed in Section 1.1, the Payment Card Industry Security Standards Council is an industry consortium of payment card issuers, including Visa and Mastercard. This body has responsibility for developing industry standards for securing cardholder data. The data security standard (DSS) takes effect whenever a merchant processes or stores cardholder data. The standard is updated every three years, and there are three versions extant, which we have modeled for our case study. Among the high level PCI-DSS requirements are goals of protecting cardholder data, securing one's network, and using access control measures.

We modeled a scenario where a football stadium was upgrading its payment card infrastructure. The stadium model captured had 44 nodes and 30 relations; the PCI-DSS had 246 goals, 297 tasks/tests, and 261 implications (connections between

goals and tasks). Our case study captured three general circumstances of change: expansion, contraction, and revision, which we focus on here.

Most of the changes to the PCI-DSS, particularly those smaller in scope, are to clarify previous requirements or to amend erroneous or irrelevant requirements. This is exactly why requirements evolution is a problem: as the standard is implemented, it becomes clear which parts are unsuited to the real-world problems of payment card security. Often, unfortunately, this evolution occurs because a hacker discovered an unanticipated hole in the security. In some sense, the standard is always one step behind these attacks. The following examples show how the standard was revised:

1. Version 1.2 of the standard required organizations to change security keys (that is, electronic keys) annually. However, in some cases it became clear that this was either too onerous, or too infrequent. Version 2.0 of the standard therefore revised this requirement to ask that organizations change keys according to best practices. Note the ambiguity in this last phrase.
2. Similarly, previous versions of the standard asked organizations to use cryptography. However, cryptography means many things, so this was updated to demand the use of strong cryptography. Consider the situation in which we (as stakeholders) had decided to use a 56-bit encryption protocol (which is easily broken). We now have to update this to a newer protocol, such as Triple-DES. This switch may conflict with our choice of technology from before, and requires us to drop support for a particular task (which would otherwise lead to an inconsistency).
3. In previous iterations of the standard, secure coding guidelines had to be followed, but only for web applications such as web pages. In the latest version, this has been revised to demand these guidelines apply to all applications. Again, this might require our IT system to change coding practices, implement testing frameworks, or hire consultants to be consistent with this revision.

We then applied the methodology described below to find solutions to these revisions in a reasonable amount of time. This is what might occur if, for example, an organization needed to understand what testing to carry out to ensure compliance: the tasks the REKB identified using GET-MIN-CHANGE-TASK-ENTAILING would correspond to the validation tests identified in the PCI-DSS standard. More information on the case study is available in [277].

1.5.2 Methodological Guidance for Solving Unanticipated Changes

Since the focus of the Requirements Evolution Problem is changing systems, it behooves us to outline the process by which these changes occur, as well as the impact the changes have on the requirements problem. Figure 1.2 outlines these steps in graphical form.

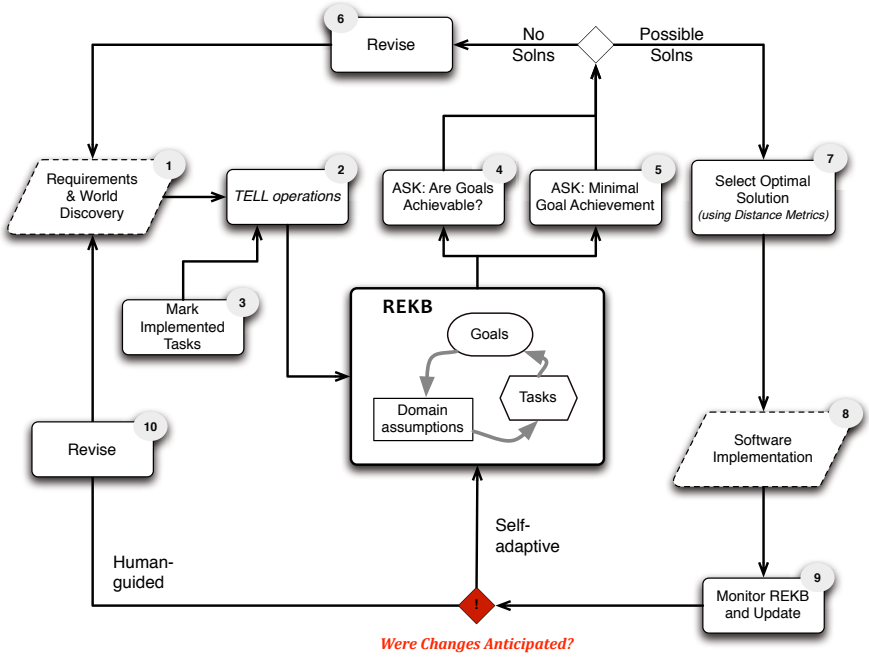

Fig. 1.2: A methodology for Requirements Evolution Problems

Step 1. Elicit requirements from stakeholders and map the speech acts into domain assumptions W, goals in R, and solution tasks in S. Define domain assumptions that are relevant to the context of the particular company. For instance, if one is working with a payment processor (like Verifone) for a 1,200 terminal soccer stadium, one will want to add the details of the Verifone-specific constraints. At the same time, identify relevant problem modules. In the case study this is the set of applicable standards and regulations: the PCI-DSS, Sarbanes-Oxley, etc. For example, requirements 1 and 1.1 of the PCI DSS could be represented as the goal G_1: "Install and maintain a firewall configuration to protect cardholder data" and goal $G_{1.1}$: "Establish firewall and router configuration standards", along with the domain assumption $W_1 : G_{1.1} \rightarrow G_1$.

Step 2. 'TELL' this requirements problem to the REKB, introducing the goals and tasks as atoms and asserting the domain assumptions.

Step 3. Identify existing implemented tasks and add to the REKB, marking them as "implemented". Rather than defining future tasks to be performed, we need to check whether the requirements problem can already be satisfied. In the first iteration, this is clearly unlikely, but in future iterations it may be possible.

Step 4. These previously implemented tasks will be the initial input for the ARE-GOALS-ACHIEVED-FROM operator. This step is essential to prevent over-analysis: if we have a set of tasks that already solve the (new) problem, just use

those. This is where the difference between adaptation (existing implementation solves the new problem) and evolution begins.

Step 5. If no candidate solutions were discovered in Step 4, then we must analyze the REKB using MINIMAL-GOAL-ACHIEVEMENT. That operation returns Σ sets of S. In the case of the PCI-DSS, this means finding (zero or more sets of) some set of tests which will satisfy the goals captured in the standard, and in particular, the goal "comply with the PCI DSS".

Step 6. If the model is not satisfiable, repeat the elicitation steps to revise the REKB. This is the process of refining our REKB in order to solve the Requirements Problem.

Step 7. Once we have Σ, which is a set of 'candidate solutions', decide on a member of Σ using decision criteria (Π). There are several possibilities, including one, maximize the number of preferred goals contained. Two, minimize a distance function between existing tasks and new tasks. Here we would make use of the previously implemented tasks for earlier versions of the system implementation.

Step 8. Implement the initial solution to the Requirements Problem as RP_1.

Step 9. Monitor the implementation, domain, and goals for changes. This can be done using e.g., awareness requirements [788].

Step 10. Something has changed (i.e. in W or G) and the system (S) can no longer satisfy our goals. We must re-evaluate the Requirements Problem to find a solution that will. We update the REKB with new information and repeat from Step 2.

The diamond with exclamation mark reflects the key distinction between a Requirements Evolution Problem and a Self-Adaptation Problem. If the detected change (step 9) was anticipated, then we can look to the current version of the REKB. Assuming the design was properly instantiated, this ought to provide a new solution from within the initial REKB. However, as in Berry et al. [100], if there is no solution in the initial REKB, we must intervene as humans and revise the REKB accordingly.

This is a high-level methodology: variants are possible, of course. For one, we could select more than one solution in Step 7 in order to maximize flexibility. Step 7 might also be expanded to reflect software product line development.

1.5.3 Revising Requirements

We mentioned that one of the key concerns in the Requirements Evolution Problem is how to manage new information which contradicts existing information. Steps 6 and 10 of the Requirements Evolution Problem methodology is predicated on revising the REKB when new information is found (assuming the REKB and its revision are consistent). We can draw on the research into belief revision in knowledge representation with a few caveats. Most importantly, it has long been argued that the context of revision is important. For example, the difference between bringing a

knowledge base up to date when the world changes (update) and revising a knowledge base with new information about a static world was outlined in Katsuno and Mendelsohn [453]. This is an important distinction in RE as well.

According to one seminal approach to belief revision, the AGM postulates [16], there are three key principles:

1. the use of "epistemic entrenchment" to partially order the formulae, in order to decide which to give up when revising the belief set;
2. the principle that the "new information" φ ought to be retained in the revised belief set;
3. information should be discarded only if necessary (minimal mutilation or information economy), because obtaining it is expensive.

The problem with these principles for the REKB is that **a)** we are dealing with three distinct sorts of well-formed formulas (wffs) (namely, goals R, specifications S and domain assumptions W_D) and **b)** our central concern is solving the requirements problem. This last point distinguishes the REKB version of revision: the concern of classical revision is the state of an agent's beliefs (e.g., that it is raining rather than sunny); the concern of requirements revision is how best to incorporate the new information in order to solve the modified requirements problem. In this formulation, the new information may in fact be rejected, whereas in AGM revision, this is never the case.

For example, consider the case where we are told that the stakeholders have a new goal: to support VISA's touchless card readers[8]. The AGM postulates would have us accept this new fact on the principle that recent information is dominant over older information. In the Requirements Evolution Problem, before accepting new information we must understand its implications with respect to solving the requirements problem. Consider the case where our soccer stadium already supports the goal of "accept touchless payment cards". If the designers are told a new customer goal is to "require signatures on payments", we can see there is a conflict, which implies the REKB must be revised (absent paraconsistent reasoning). The AGM postulates would say that the new goal is paramount, and that the old goal be rejected (or a workaround devised). In a design situation, however, this new goal may be illogical, and should itself be rejected. In this situation the best we can do is ask for preferences between these conflicting goals. We reject it not because it imperils the current solution, but because it conflicts with other goals in the sense that we cannot solve them simultaneously.

This leads to a new definition of revision in the REKB formulation of the requirements problem. When domain assumptions change, since these are invariant by definition, we apply standard belief revision operators to those wffs. For example, if previously we had believed that "50% of the clientele possess touchless credit cards", and after monitoring sales for a few months, our statistics inform us that the figure is closer to "90%", it seems intuitive to accept the new information. In this case, our domain assumptions are ordered using an epistemic entrenchment relation.

[8] A touchless card reader is referred to as PayPass or PayWave, and does not require a swipe or insertion for low-value transactions.

For goals and specifications, we have broad freedom to reject changes. Our motivation for deciding on the particular revision to accept is with respect to the requirements problem. We prefer a new state of the REKB that brings us better solutions. The definition of 'better' solution will be defined with respect to the distance function we use in GET-MIN-CHANGE-TASK-ENTAILING, i.e. Π. This means that even if stakeholders inform us of new tasks that have been implemented, or new goals they desire, we may reject these revisions if they do not lead to a better solution. This might be the case if, as with the previous example, stakeholders tell us that they have upgraded the payment terminals to accept touchless payments. It may be that this is now possible, but still does not satisfy other goals in our REKB. This ability to reject the most current revision, unlike classical belief revision, means that revising requirements problems is properly aligned with our definition of the REKB as a support mechanism for design decisions.

1.5.4 Selecting Non-Dominated Solutions

The second challenge in requirements evolution was deciding what solutions to select when the REKB has changed and been revised. Recall the GET-MIN-CHANGE-TASK-ENTAILING operator takes a set of goals and a set S_0 of tasks, the old implementation, and returns a set of sets of tasks Σ which are equally desirable (non-dominated) solutions to the requirements problem with respect to a distance function. The important consideration in choosing new solutions is the choice of a distance function (i.e., Π above), so let us examine some possible choices.

Requirements re-use is important, so we do not want to completely ignore previous implementations in selecting a new solution. That suggests there are properties of a new solution with respect to the old one that might be useful heuristics for the decision. We defined several properties Π in [277], together with illustrative examples based on a case where: $S_0 = \{a,b,c,d,e\}$ was the initial solution (the set of tasks that were implemented); and $S_1 = \{f,g,h\}, S_2 = \{a,c,d,f\}$ and $S_3 = \{a,b,c,d,f\}$ are minimal sets of tasks identified as solutions to the new requirements:

1. *The standard solutions*: this option ignores the fact that the new problem was obtained by evolution, and looks for solutions in the standard way. In the example, one might return all the possible new solutions $\{S_1, S_2, S_3\}$, or just the minimum size one, S_1.
2. *Minimal change effort solutions*: These approaches look for solutions $\hat{S}$ that minimize the extra effort $\hat{S} - S_0$ required to implement the new "machine" (specification). In our view of solutions as sets of tasks, $\hat{S} - S_0$ may be taken as "set subtraction", in which case one might look for (i) the smallest difference cardinality $| \hat{S} - S_0 |$ (S_2 or S_3 each requires only one new task to be added/implemented on top of what is in S_0); or (ii) smallest difference cardinality *and* least size $| \hat{S} |$ (S_2 in this case).
3. *Maximal familiarity solutions*: These approaches look for solutions $\hat{S}$ that maximize the set of tasks used in the current solution, $\hat{S} \cap S_0$. One might prefer such

an approach because it preserves most of the structure of the current solution, and hence maximizes familiarity to users and maintainers alike. In the above example, S_3 would be the choice here.

4. *Solution reuse over history of changes*: Since the software has probably undergone a series of changes, each resulting in newly implemented task sets $S_0^1, S_0^2, ..., S_0^n$, one can try to maximize reuse of these (and thereby even further minimize current extra effort) by using $\bigcup_j S_0^j$ instead of S_0 in the earlier proposals.

The above list makes it clear that there is unlikely to be a single optimal answer, and that once again the best to expect is to support the analyst in exploring alternatives.

1.5.5 Summary

This framework supports the iterative 'exploration' of one's requirements, domain knowledge, and solution. As analysts, one can ASK questions of the REKB and understand how complete or accurate the solution will be. Furthermore, using GET-MIN-CHANGE-TASK-ENTAILING, iterating and incrementing the solution itself, particularly in response to change, happens continuously, as new information is added to the REKB. It focuses on requirements as first-class citizens of software evolution and tries to reconcile the satisfaction of those requirements by a suitable software specification, respecting domain knowledge.

In addition to the methodology, we also need to track and version our artifacts using metaphors from version control (e.g., diff, checkin, etc.). We would also like our REKB to be scalable to models of reasonable size: in a related paper [277], we showed incremental reasoning was possible 'online', i.e., in less than 10 seconds.

Related work includes:

- The 'cone of uncertainty', which captures the idea that before the project begins uncertainty about exactly what the requirements are is quite broad. The cone narrows as the project progresses and we have more information about the relevant requirements.
- Levels of knowledge, including 'known knowns', 'known unknowns' (changes we anticipate), 'unknown knowns', or tacit knowledge, and 'unknown unknowns', possible changes we are not aware of. These illustrate the major challenge in requirements evolution. However, being aware of these levels of knowledge is already a major achievement in most projects.
- The Architecture Tradeoff Analysis Method (ATAM), which incorporates analysis of sensitivity points and trade-offs against non-functional quality attributes of the software. This is a scenario exploration technique designed to test a possible design against changes.
- The V-model (and all other related testing approaches) which insists that requirements are reviewed early by the test team in order to ensure testability.

1.6 Conclusions

In this chapter, we have made the point that focusing solely on implementation artifacts is insufficient. It is the requirements which are providing the guidance for maximizing stakeholder value, and so understanding, modeling, and reasoning about evolving requirements is extremely important. We discussed how research in software evolution led to research in requirements evolution, and showcased some of the industrial and academic approaches to managing requirements evolution. The previous section on the REKB defined our approach to the requirements evolution problem: that of incremental exploration of the problem space using the REKB as a form of workbench. Since we expect our system to be subject to change pressures, and constantly evolving, supporting exploration allows both an initial problem exploration, as well as a revision when something previously established changes.

The vision for managing the Requirements Evolution Problem is growing closer to the vision of adaptive software systems (Chapter 7.6). In both cases, we would like to support rapid and assured changes to a software system, and in the adaptive case, without human intervention. To date, the primary difference is in which artifacts are in focus. For requirements at run-time, the requirements model is viewed as the driver of system understanding. At runtime we need to monitor how the system is doing with respect to its requirements. This is best done by comparing the execution (trace) to a runtime version of requirements, rather than a runtime model of the implementation. The implementation is responsible for the actual system. However, in answering questions about how the system is performing, e.g. with respect to quality attributes or high-level goals, one can only answer these questions (directly anyway) by understanding the state of execution of requirements. Requirements evolution is a complex problem, but supporting incremental and iterative analysis of the requirements model will help us in making software in general more adaptable and efficient.