How requirements evolution influences software evolution

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

Abstract 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, especially with respect to accommodating legal and nonfunctional requirements. 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 existing approaches to requirements evolution with respect to these two problems. Finally, the chapter describes a framework for supporting requirements evolution, and discusses a real-life case study of the payment card industry.[1]

Neil Ernst
University of British Columbia, e-mail: `nernst@cs.ubc.ca`

Alexander Borgida
Rutgers University, e-mail: `borgida@cs.rutgers.edu`

Ivan J. Jureta
University of Namur, e-mail: `ivan.jureta@fundp.ac.be`

John Mylopoulos
University of Trento, e-mail: `jm@disi.unitn.it`

[1] Portions of this chapter are adopted from [28].

1 The importance of requirements in software evolution

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 one's requirements to be constant and unchanging 'north stars' is a recipe for disaster. The 'big design up front' approach is no longer defensible, particularly in a business environment that emphasizes speed and resiliency to change [39].

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 2. In 2005, a paper on "challenges in software evolution" [51] 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 [50] 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, most change tasks involve low-level corrective maintenance tasks, rather than more complex adaptive maintenance. While the numbers of change tasks might be greater, our position is that the adaptive changes implied by requirements evolution are more complex and more costly, and therefore more important, than identifying bug fixes. Finally, requirements change is seen as part of the problem domain and therefore untouchable, much like understanding the organizational objectives might be. We believe that the transition from problem to solution is of paramount importance in software development.

It is our view that requirements documents should drive implementation decisions. In other words, requirements must be tangible, and requirements must be relevant. While they may take the form of work item lists, as is the case in most industrial tools, preferably they would 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., [29]) where the key challenge is 're-quirements repair': re-evaluating the available solutions to solve the changed requirements, adding (minimal) new implementations where it is necessary [52].

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.

1.1 The Requirements Problem

As a reference framework, we introduce an ontology for requirements. This work is based on [41] and [40], both of which derive from the fundamental requirements problem of Zave and Jackson (Z&J) [73]. 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, or RML. In Z&J's original formulation, given a set of *requirements* R (which are optative—expressing desires), the requirements problem is to find a *system specification* S and (indicative) *world knowledge* W that holds such that $W \cup S \vdash R$ but $W \cup S \not\vdash \bot$ ($\vdash$ being classical logical deduction and $\bot$).

The **Requirements Evolution Problem** extends this notion to introduce change over time:

> **Problem statement:** Given (i) requirements R, domain knowledge W, and (ii) some chosen *existing* solution S_0 of tasks (i.e., one that satisfies $W, S_0 \vdash R$), as well as (iii) modified requirements problem ($\delta(\text{R}), \delta(\text{W}), \delta(\text{T})$) that include modified requirements, domain knowledge and possible tasks, produce a subset of possible specifications $\hat{S}$ to the changed requirements problem (i.e., $\delta(W), \hat{S} \vdash \delta(R)$) which satisfy some desired property Π, relating $\hat{S}$ to S_0 and possibly other aspects of the changes.

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 impacts the validity of the inferences drawn. For example, changes in requirements R, i.e., from R_0

to R_1, will force a re-evaluation of whether $W \cup S$ still classically entails the satisfaction of R_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 [45]. These papers characterize the nature of software evolution as an 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 as a consequence of such maintenance 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 [12, 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 (40% by some measures [11]). Swanson [61] 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 *adaptive* maintenance activity. Swanson's paper marks one of the first times researchers realized that it wasn't 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 wasn't 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 [56] 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" [56, 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 [49, p. 42].

In response to the problems with the waterfall approach, iterative models, such as Boehm's 'spiral' model of development [14] 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 – a very recent emphasis in software engineering (e.g., [4]). 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." [16] The point of Brooks's quote is to emphasize how little one can anticipate the real concerns in designing software systems, particularly novel, complex 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 [22].

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, if 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 [18, p. 17] concludes that evolution only occurs when maintenance impacts business rules or changes properties visible to the customer.

Harker et al. [33] extended Swanson's work to focus on change drivers with respect to system requirements (summarized in Table 1), because "changing

requirements, rather than stable ones, are the norm in systems development."
[33, 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 Types of requirements change [33]

	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
[15]. He introduces the notion of shearing layers for buildings, which distin-
guish 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 re-
quirements is that good design ought to identify which requirements are more
change-prone than others, and structure a solution based on that assump-
tion. 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 "Con-
nect 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 [42]. They empha-
sized that requirements evolution is inevitable and must be managed by pay-
ing 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 [11]. Somerville and Sawyer's requirements text-

book [60] 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 research projects in the area of information systems modeling have touched on evolution. CIM [17] 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 [17, p.401]." Little guidance was given on how to do this, however. In a similar vein, RML [32], ERAE [25] and Telos [53] 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 [26]. We return to the use of formal logic for managing evolving requirements in Section 4.

Finally, one could consider the elaboration of a requirements model (e.g., from high-level objectives to lower-level technical requirements) as 'evolving' requirements (as in [8]); we focus on requirements models for which elicitation is assumed to be completed, and now changes, rather than the process of requirements elicitation at an intermediate point in time.

2.2 Empirical Studies of Requirements Evolution

This section motivates the importance of the Requirements Evolution Problem (REP) by reviewing several academic studies that address the issue. In particular, the focus of this section is on research projects which looked at industrial REPs. Many industrial case studies focus on source code evolution, and little attention is paid to the requirements themselves (which presumably are driving a number of changes to source code). This is likely because requirements are often not available explicitly, unlike source code. This is particularly true in the case of open-source software studies. 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 [3],

which includes updates since the initial release in 1972, Chapter 9 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").

Chung et al. [21] 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 particular, the paper focused on tracking the impact of a change on other non-functional properties of the system, such as accuracy and timeliness. Their notation allows analysts to model priorities and perform semi-automated analysis of the updated model. The paper concludes with some principles for accommodating change.

In [9], 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.

Anderson and colleagues conducted a series of case studies of changing requirements [6, 7]. 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.

Tun et al [63] use automated extraction techniques to obtain problem structures from a large-scale open source project (Vim). 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. 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 [35]. They correlated project specifications to source code commits and were able to identify related changes.

Many studies of changing requirements have focused on software product lines. In a product line environment, one has many different products which are assembled from varying combinations of implementation level artifacts. An example of a product line is car models: one might have a base model with no niceties such as power steering, an intermediate model with improved options, and a top-end model with a bigger engine, more audio capability, and so on. The key insight is that the underlying car is the same in all cases.

Change is clearly a concern; does a change get implemented for all products? Which products contain that requirement? Industrial case studies looking at this issue include [69] and [70].

Herrmann et al [34] 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 issue of changing requirements in the highly formal requirements environment of spacecraft design was considered in [54], 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 to minimize the dependencies between abstraction levels.

A final example industrial case study [48] designed a framework for evolving security requirements and applied it to the air traffic control domain. They first used interviews with stakeholders (such as controllers) to model the security requirements of the domain, then asked users to conduct an evolution scenario and identify what must change in the existing system. While preliminary, this shows the practical benefits possible by focusing on requirements.

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.

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., [68, 44]) 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.

3.1 Standards and Industry

IEEE Standard 830 [37], 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. The standard acknowledges that evolutionary revisions may be inevitable.

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 is hierarchical lists (e.g., "R4.2.4 the system shall ..."). Traceability is a big feature, and requirements can be linked (to one another and to other artifacts, such as UML diagrams). Multiple users are supported, and changes 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.

---

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

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

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 [5] has made popular visual work-in-progress displays, the most basic of which are whiteboards with lifecycle 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.

3.4 Summary

Particularly for smaller organizations, requirements are not treated at a high-level, often existing as an Excel spreadsheet or maintained implicitly [10]. 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

---

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

[5] http://agilemanifesto.org/

Collaboration (OSLC)[6] is one initiative that looks to overcome the traditional silos.

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 post-implementation to drive changes in the software. Interest in the notion of requirements at run-time has greatly increased recently, however, and we touch on this below. There are definite overlaps with work on adaptive software (see the chapter later in this book by Müller et al.) and model-driven evolution.

4.1 Problem Frames Approach

We mentioned the empirical study of Tun et al. [63] earlier. This work builds on the seminal problem frames approach of Michael Jackson [38] 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 Z&J 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".

Another project by Tun et al.[62] 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. Finally, Yu et al. [72] 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.

---

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

4.2 Extensions of the NFR Framework

The NFR model, introduced in [20], 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. [31] and Sebastiani et al. [59] 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.

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., [47], [43]) allows these qualitative models to capture context-driven variability, a point also made in Ali et al. [2], who make the case that requirements variability necessitates the monitoring of the operating contexts. This monitoring 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. [23] 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.

4.3 Run-time Adaptive Requirements

Work in the area of adaptive requirements is tangentially connected. In particular, the notion of "requirements at runtime", explored in a series of workshops at the Requirements Engineering conference (see appendix) , introduced the notion of using requirements models to drive system changes. Qureshi et al. [58] 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. In the RELAX framework [67], a language is designed to specifically manage "the explicit expression of environmental uncertainty in requirements". When a new situation is monitored in W (the world), the RELAX language is looked to for ways to loosen the specification S. In similar fashion, Epifani et al. [27] 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.

---

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

4.4 KAOS-based Approaches

A major contribution to the RE literature was the KAOS goal modeling framework, first introduced in [24]. This 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 [65]). Second, variants can be managed as a form of product line, and called upon if and when things change (see below for more on product lines and requirements). Later work [46] 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.

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 work described above). Here we review two approaches.

Default logic approaches, appearing in [74] and [30], rely on David Poole's Theorist system [57] 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 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, the original sentence cannot be retracted.

Closely aligned with this perspective is the REFORM framework [30], 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 et al. also define 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.

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 is then found to be false. For example, we might say at time T_0 we must satisfy a requirement R_x, but at a later time, T_n, change our minds (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. [55]:

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

Hunter and Nuseibeh [36] 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.

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

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 [19], 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 can then be used to extract interesting labels for that data, including which quality requirements are affected, as in Hindle's work. 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. [34] 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 $R_1 \setminus R_0$). The challenge with this approach is to correctly characterize the existing system's initial requirements.

Welsh and Sawyer [66] 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)
- broken assumption (an incorrect fact in W)
- new technology (new elements in S)
- consequential change (changes to the inferences drawn from $W, S \vdash R$)
- 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 [71] to annotate the models with possible changes and impacts. The primary contribution is to support elicitation and modelling.

4.7 Feature Models

Feature models are covered in greater detail in Chapter 3. 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 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.

4.8 Summary

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. 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 [66], Ghose [30], and several others focus on understanding the nature of the problem using classification techniques.

One of the seminal papers in characterizing evolutionary systems is that of Berry et al. [13]. 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. [27] or Whittle et al. [67].

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. [23].

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.

5 A Framework for Requirements Evolution

This section considers implementation-level strategies for managing Requirements Evolution Problems (REP). In our work, we represent requirements as goals G according to goal-oriented requirements engineering [64]. With that in mind, let us recall the definition of the Requirements Evolution Problem:

> **Problem statement:** Given (i) goals G, domain knowledge W, and (ii) some chosen *existing* solution S_0 of tasks (i.e., one that satisfies $W, S_0 \vdash G$), as well as (iii) modified requirements $(\delta(G), \delta(W), \delta(T))$ that include modified goals, domain knowledge and possible tasks, produce a subset of possible specifications $\hat{S}$ to the changed requirements problem (i.e., $\delta(W), \hat{S} \vdash \delta(G)$) which satisfy some desired property Π, relating $\hat{S}$ to S_0 and possibly other aspects of the changes.

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. REKB knows about items stored within it, and that knowledge is leveraged using operations. In [29] 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 a set of tasks in
  REKB can satisfy a given set of goals. This is the "forward reasoning"
  problem of Giorgini et al. [31].
MINIMAL-GOAL-ACHIEVEMENT   The opposite of the first operation, REKB
  discovers sets of tasks (if any) which minimally satisfy desired goals. This
  is the (abductive) backward reasoning problem of Sebastiani et al. [59].
GET-MIN-CHANGE-TASK-ENTAILING   This operation produces solutions
  (if any) to the REP. Given an initial solution S and a set of desired goals,
  find minimal sets of tasks which are non-dominated for some criterion of
  minimality and satisfy the new requirements problem. The key idea 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 'unantici-
pated' we mean that there exists no mechanism in the implementation speci-
fication S to accommodate these changes. This clearly positions the Require-
ments Evolution Problem as distinct from the Self Adaptation Problem. The
Self Adaptation Problem (SAP) is quite different from the REP. With re-
spect to the aspects of G, W, and S, the SAP is to accommodate changes
in D,G by creating a suitably adaptive $\hat{S}$ *ab initio*. In other words, unlike
the REP, self-adaptation approaches do not modify the implementation. It is
becoming clear, however, 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.
    There are two key concerns in the REP:

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.

5.1 The Payment Card Industry Example

The Payment Card Industry Security Standards Council is an industry con-
sortium 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 pro-
cesses 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. The 1.2 version 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. The 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. Cryptography means many things, however, 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 [29].

5.2 Methodological Guidance for Solving Unanticipated Changes

Since the focus of the REP 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. Fig. 1 outlines these steps in graphical form.
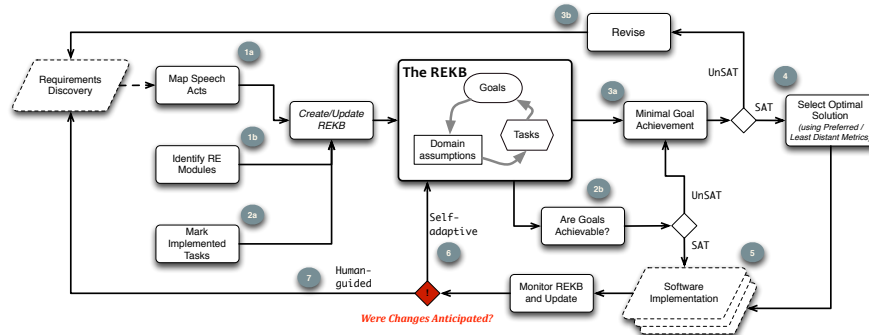

Fig. 1 A methodology for Requirements Evolution Problems

Step 1a. Elicit requirements from stakeholders and map the speech acts into domain assumptions, goals, tasks, and attitudes. Define domain assumptions that are relevant to the context of the particular company. For instance, if one is working with payment processor (like Verifone) for the 1,200 terminal soccer stadium, one will want to add the details of the Verifone-specific constraints.

Step 1b. 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, the requirements 1 and 1.1 of the PCI DSS could be represented as the mandatory 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 $K_1 : G_{1.1} \rightarrow G_1$.

Step 2a. 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 may be possible.

Step 2b. These previously implemented tasks will be the initial input into 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 3a. If no candidate solutions were discovered in Step 2, then we must analyze the REKB using MINIMAL-GOAL-ACHIEVEMENT. We are presented with sets of tasks S. In the case of the PCI-DSS, this means finding the sets of tests which will satisfy the mandatory goals (and in particular, the goal "comply with PCI DSS").

Step 3b. 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 4. Once we have a candidate solution, decide on a member of S using decision criteria. There are two sets of criteria: one optimizes the static criteria, i.e., maximize the number of preferred goals contained. The other set 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 5. Implement the initial solution to the Requirements Problem as RP_1.

Step 6. Monitor the implementation, domain, and goals for changes. This can be done using e.g., requirements monitors.

Step 7. Something has changed and the system can no longer satisfy our mandatory goals. We must re-evaluate the Requirements Problem to find a solution that will (RP_2). Update the REKB and repeat from Step 2.

The diamond with exclamation mark reflects the key distinction between a REP and a Self-Adaptation Problem. If the detected change (step 6) 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 with Berry et al. [13], 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 4 in order to maximize flexibility. Step 4 might also be expanded to reflect software product line development.

5.3 Revising Requirements

We mentioned that one of the key concerns in the REP is how to manage new information which contradicts existing information. Step 2a of the REP methodology is predicated on revising the REKB when new information is found (assuming the REKB and its revision are consistent). The concept of belief revision in artificial intelligence, especially the so-called AGM principles [1], closely parallels the concept of REKB revision. And yet, there are important distinctions that mean that we do not accept the AI belief revision literature in its entirety for the Requirements Evolution Problem.

There are three key principles in belief revision.

1. The use of epistemic entrenchment to partially order the formulae, in order to decide which to do away with when revising the belief set;
2. The principle that the "new information" ϕ ought to be retained in the revised belief set;
3. Gathering/learning information is expensive, and so should be discarded only if necessary (minimal mutilation or information economy).

The problem with these principles for the REKB is that **a)** we are dealing with three distinct SORTs of wffs (namely, goals, tasks and domain assumptions) and **b)** our central concern is solving the requirements problem. This last criteria distinguishes this type of revision: the concern for classical revision is the state of an agent's beliefs (e.g., that it is raining rather than sunny). In the requirements problem, however, the concern is how best to incorporate the new information in order to solve the revised requirements problem. In this formulation, the new information may in fact be rejected, whereas in AGM revision, this is never the case.

Consider the case where we as designers are told that the stakeholders have a new goal to support VISA's touchless card readers[8]. The AGM belief revision postulates would have us accept this new fact on principle (using the notion of recent information being dominant over older information). This is intuitive in the knowledge representation problem, where we are dealing with facts in the world. But in the design problem, preferring recent information is not always the correct approach. Before accepting the new information we must understand the implications, with respect to solving the requirements problem, of acceptance. Consider the case where a meeting scheduler already supports the goal of "managing schedules automatically". If the designers are told a new customer goal is to "allow users to enter information in a paper organizer", we can see there is a conflict, which implies the REKB must be revised (absent paraconsistent reasoning). AGM postulates would say that the new goal is paramount, and that the old goal is 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. Nonetheless, 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 tasks, 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. 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.

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 (the mandatory goals) and a set S of tasks, the old implementation, and returns a set of sets of tasks 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 distance function, so let us examine some possible choices.

section could probably be spiced up with case study examples.

Requirements re-use is important, so we do not want to completely ignore previous implementations. So what criteria are important in selecting this new solution? We defined several properties Π in [29], 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 require 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.

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.

In addition to the methodology, we also need to track and version our artifacts using version control like metaphors (e.g., diff, checkin, etc.). We would also like our REKB to be scalable to models of reasonable size: in a related paper [29], we showed incremental reasoning was possible 'online', i.e., in less than 10 seconds.

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 REP is increasingly growing closer to the vision of adaptive software systems. In both cases, we would like to support rapid and assured changes to a software system, preferably 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.

References

1. ALCHOURRÓN, C. E., GÄRDENFORS, P., AND MAKINSON, D. On the Logic of Theory Change: Partial Meet Contraction and Revision Functions. *Journal of Symbolic Logic 50*, 2 (1985), 510–530.

2. ALI, R., DALPIAZ, F., GIORGINI, P., AND SOUZA, V. E. S. Requirements Evolution: From Assumptions to Reality. In *International Conference on Exploring Modeling Methods in Systems Analysis and Design* (London, 2011), pp. 1–10.

3. ALSPAUGH, T. A., FAULK, S. R., BRITTON, K. H., PARKER, R. A., PARNAS, D. L., AND SHORE, J. E. Software Requirements for the A-7E Aircraft. Tech. rep., 1992.

4. AMBLER, S. W. Examining the "Big Requirements Up Front (BRUF)" Approach. Tech. rep., 2006.

5. ANDERSON, D. J. *Kanban*. Blue Hole Press, 2010.

6. ANDERSON, S., AND FELICI, M. Controlling Requirements Evolution: An Avionics Case Study. In *International Conference on Computer Safety, Reliability and Security* (Rotterdam, 2000), F. Koornneef and M. van Der Meulen, Eds., p. 0.

7. ANDERSON, S., AND FELICI, M. Requirements Evolution: From Process to Product Oriented Management. In *International Conference on Product Focused Software Process Improvement* (Kaiserslautern, Germany, 2001), pp. 27–41.

8. ANTÓN, A. I. Goal-Based Requirements Analysis. In *Int. Conf. on Req. Engineering* (Colorado Springs, Colorado, 1996), pp. 136–144.

9. ANTÓN, A. I., AND POTTS, C. Functional paleontology: system evolution as the user sees it. In *Int. Conf. on Software Engineering* (Toronto, Canada, 2001), pp. 421–430.

10. ARANDA, J., EASTERBROOK, S. M., AND WILSON, G. V. Requirements in the wild: How small companies do it. In *Int. Conf. on Req. Engineering* (Delhi, India, 2007).

11. BASILI, V., AND PERRICONE, B. T. Software Errors and Complexity: An Empirical Investigation. *Commun. ACM 27* (1984), 42–52.

12. BELADY, L. A., AND LEHMAN, M. M. A model of large program development. *IBM Systems Journal 3* (1976), 225–252.

13. BERRY, D. M., CHENG, B. H. C., AND ZHANG, J. The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems. In *International Conference on Requirements Engineering: Foundation for Software Quality* (Porto, Portugal, 2005), pp. 113–120.

14. BOEHM, B. A Spiral Model of Software Development and Enhancement. *IEEE Computer 21*, 5 (1988), 61–72.

15. BRAND, S. *How Buildings Learn: What Happens After They're Built*. Viking Press, 1995.

16. BROOKS, F. P. *The mythical man-month*, 1st ed. Addison Wesley, Reading, Mass., 1975.

17. BUBENKO, J. A. Information modeling in the context of system development. In *IFIP Congress* (1980), pp. 395–411.

18. CHAPIN, N., HALE, J. E., FERNANDEZ-RAMIL, J. C., AND TAN, W.-G. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice 13*, 1 (2001), 3–30.

19. CHARRADA, E. B., KOZIOLEK, A., AND GLINZ, M. Identifying outdated requirements based on source code changes. In *Int. Conf. on Req. Engineering* (Chicago, IL, June 2012).

20. CHUNG, L., MYLOPOULOS, J., AND NIXON, B. A. Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. *Trans. Soft. Eng. 18* (1992), 483–497.

21. CHUNG, L., NIXON, B. A., AND YU, E. S. Dealing with change: An approach using non-functional requirements. *Requirements Engineering J. 1*, 4 (1996), 238–260.

22. COCKBURN, A. Using both incremental and iterative development. *STSC CrossTalk 21*, 5 (2008), 27–30.

23. DALPIAZ, F., GIORGINI, P., AND MYLOPOULOS, J. An Architecture for Requirements-Driven Self-reconfiguration. In *International Conference Advanced Informations Systems Engineering* (Amsterdam, 2009), pp. 246–260.

24. DARDENNE, A., VAN LAMSWEERDE, A., AND FICKAS, S. Goal-directed requirements acquisition. *Science of Computer Programming 20*, 1-2 (1993), 3–50.

25. DUBOIS, E., HAGELSTEIN, J., LAHOU, E., PONSAERT, F., AND RIFAUT, A. A knowledge representation language for requirements engineering. *Proceedings of the IEEE 74* (1986), 1431–1444.

26. EASTERBROOK, S. M., AND NUSEIBEH, B. A. Managing inconsistencies in an evolving specification. In *Int. Conf. on Req. Engineering* (York, England, 1995), pp. 48–55.

27. EPIFANI, I., GHEZZI, C., MIRANDOLA, R., AND TAMBURRELLI, G. Model evolution by run-time parameter adaptation. *International Conference on Software Engineering* (2009), 111–121.

28. ERNST, N. *Software Evolution: A Requirements Engineering Approach*. PhD thesis, University of Toronto, 2012.

29. ERNST, N., BORGIDA, A., AND JURETA, I. Finding Incremental Solutions for Evolving Requirements. In *Int. Conf. on Req. Engineering* (Trento, Italy, Feb. 2011), pp. 15–24.

30. GHOSE, A. K. Formal tools for managing inconsistency and change in RE. In *International Workshop on Software Specification and Design* (2000), pp. 171–181.

31. GIORGINI, P., MYLOPOULOS, J., NICCHIARELLI, E., AND SEBASTIANI, R. Formal Reasoning Techniques for Goal Models. *Journal on Data Semantics 2800* (2003), 1–20.

32. GREENSPAN, S., MYLOPOULOS, J., AND BORGIDA, A. Capturing more world knowledge in the requirements specification. In *Int. Conf. on Software Engineering* (Tokyo, 1982), pp. 225–234.

33. HARKER, S. D. P., EASON, K. D., AND DOBSON, J. E. The change and evolution of requirements as a challenge to the practice of software engineering. In *Int. Conf. on Req. Engineering* (1993), pp. 266–272.

34. HERRMANN, A., WALLNÖFER, A., AND PAECH, B. Specifying Changes Only – A Case Study on Delta Requirements . In *International Conference on Requirements Engineering: Foundation for Software Quality* (Amsterdam, Apr. 2009), pp. 45–58.

35. Hindle, A., Bird, C., Zimmermann, T., and Nagappan, N. Relating Requirements to Implementation via Topic Analysis: Do Topics Extracted from Requirements Make Sense to Managers and Developers? In *International Conference on Software Maintenance* (Trento, Italy, 2012), pp. 1–12.

36. Hunter, A., and Nuseibeh, B. A. Managing inconsistent specifications: reasoning, analysis, and action. *ACM Transactions on Software Engineering and Methodology 7*, 4 (1998).

37. IEEE Software Engineering Standards Committee. IEEE Recommended Practice for Software Requirements Specifications. Tech. rep., 1998.

38. Jackson, M. *Problem Frames: Analysing & Structuring Software Development Problems.* 2000.

39. Jarke, M., Loucopoulos, P., Lyytinen, K., Mylopoulos, J., and Robinson, W. N. The Brave New World of Design Requirements: Four Key Principles. In *International Conference Advanced Informations Systems Engineering* (Hammaret, Tunisia, 2010), pp. 470–482.

40. Jureta, I. J., Borgida, A., Ernst, N., and Mylopoulos, J. Techne: Towards a New Generation of Requirements Modeling Languages with Goals, Preferences, and Inconsistency Handling. In *Int. Conf. on Req. Engineering* (Sydney, Australia, 2010), pp. 115–124.

41. Jureta, I. J., Mylopoulos, J., and Faulkner, S. Revisiting the Core Ontology and Problem in Requirements Engineering. In *Int. Conf. on Req. Engineering* (Barcelona, 2008), pp. 71—-80.

42. Lam, W., and Loomes, M. Requirements Evolution in the Midst of Environmental Change: A Managed Approach. In *Euromicro Conference on Software Maintenance and Reengineering* (Florence, Italy, 1998), pp. 121–127.

43. Lapouchnian, A., and Mylopoulos, J. Modeling Domain Variability in Requirements Engineering with Contexts. In *International Conference on Conceptual Modelling* (Gramado, Brazil, 2009), pp. 115–130.

44. Leffingwell, D., and Widrig, D. *Managing Software Requirements: A Use Case Approach*, 2nd ed. Addison-Wesley Professional, 2003.

45. Lehman, M. M., and Fernandez-Ramil, J. C. Software evolution. In *Software Evolution and Feedback: Theory and Practice*, N. H. Madhavji, J. C. Fernandez-Ramil, and D. E. Perry, Eds. Wiley, 2006, pp. 7–40.

46. Letier, E., and van Lamsweerde, A. Reasoning about partial goal satisfaction for requirements and design engineering. In *International Conference on Foundations of Software Engineering* (Newport Beach, CA, 2004), ACM Press, pp. 53—-62.

47. Liaskos, S., Lapouchnian, A., Yu, Y., Yu, E. S., and Mylopoulos, J. On Goal-based Variability Acquisition and Analysis. In *Int. Conf. on Req. Engineering* (Minneapolis, Minnesota, 2006).

48. Massacci, F., Nagaraj, D., Paci, F., Tran, L. M. S., and Tedeschi, A. Assessing a requirements evolution approach: Empirical studies in the Air Traffic Management domain. In *Empirical RE workshop at RE* (Chicago, 2012), pp. 49–56.

49. McDonald, C. From Art Form to Engineering Discipline?: A History of US Military Software Development Standards, 1974-1998. *IEEE Annals of the History of Computing 32*, 4 (2010), 32–45.

50. Mens, T. Future Research Challenges in Software Evolution. In *Presentation to ERCIM Working Group on Software Evolution* (Brussels, Sept. 2009), pp. 1–17.

51. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., and Jazayeri, M. Challenges in Software Evolution. *Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution* (Dec. 2005), 13–22.

52. Mylopoulos, J. The Requirements Problem Revisited. In *Presentation to IFIP Working Group 2.9* (Cancun, 2011).

53. Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M. Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems 8* (1990), 325–362.

54. Navarro, I., Leveson, N., and Lunqvist, K. Semantic decoupling: reducing the impact of requirement changes. *Requirements Engineering J. 15*, 4 (2010), 419–437.

55. Nuseibeh, B. A., Easterbrook, S. M., and Russo, A. Making inconsistency respectable in software development. *Journal of Systems and Software 58*, 2 (2001), 171–180.

56. Parnas, D. L. Software Aspects of Strategic Defense Systems. *Commun. ACM 28* (1985), 1326–1335.

57. Poole, D. A logical framework for default reasoning. *Artificial Intelligence 36*, 1 (1988), 27–47.

58. Qureshi, N. A., Jureta, I. J., and Perini, A. Requirements Engineering for Self-Adaptive Systems : Core Ontology and Problem Statement. In *International Conference Advanced Informations Systems Engineering* (London, 2011), pp. 1–15.

59. Sebastiani, R., Giorgini, P., and Mylopoulos, J. Simple and Minimum-Cost Satisfiability for Goal Models. In *CAISE* (Riga, Latvia, 2004), pp. 20–35.

60. Sommerville, I., and Sawyer, P. *Requirements Engineering: A Good Practice Guide*. Wiley, New York, NY, USA, 1997.

61. Swanson, E. B. The dimensions of maintenance. In *Int. Conf. on Software Engineering* (San Francisco, California, 1976), pp. 492–497.

62. Tun, T. T., Trew, T., Jackson, M., Laney, R., and Nuseibeh, B. A. Specifying features of an evolving software system. *Software - Practice and Experience 39*, 11 (2009), 973–1002.

63. Tun, T. T., Yu, Y., Laney, R., and Nuseibeh, B. A. Recovering Problem Structures to Support the Evolution of Software Systems. Tech. rep., Milton Keynes, 2008.

64. van Lamsweerde, A. Goal-Oriented Requirements Engineering: A Guided Tour. In *Int. Conf. on Req. Engineering* (Toronto, 2001), pp. 249–263.

65. van Lamsweerde, A., and Letier, E. Handling obstacles in goal-oriented requirements engineering. *Trans. Soft. Eng. 26* (2000), 978–1005.

66. Welsh, K., and Sawyer, P. Requirements Tracing to Support Change in Dynamically Adaptive Systems. *International Conference on Requirements Engineering: Foundation for Software Quality* (Apr. 2009), 59–73.

67. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B. H. C., and Bruel, J.-M. RE-LAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *Int. Conf. on Req. Engineering* (Atlanta, 2009), pp. 79–88.

68. Wiegers, K. E. *Software Requirements*. Microsoft Press, 2003.

69. Wnuk, K., Regnell, B., and Karlsson, L. What Happened to Our Features? Visualization and Understanding of Scope Change Dynamics in a Large-Scale Industrial Setting. In *Int. Conf. on Req. Engineering* (2009), IEEE, pp. 89–98.

70. Xue, Y., Xing, Z., and Jarzabek, S. Understanding Feature Evolution in a Family of Product Variants. In *Working Conference on Reverse Engineering* (2010), IEEE, pp. 109–118.

71. Yu, E. S. Towards modelling and reasoning support for early-phase requirements engineering. In *Int. Conf. on Req. Engineering* (Annapolis, Maryland, 1997), pp. 226–235.

72. Yu, Y., Tun, T. T., Tedeschi, A., Franqueira, V. N. L., and Nuseibeh, B. A. OpenArgue: Supporting argumentation to evolve secure software systems. In *Int. Conf. on Req. Engineering* (Trento, 2011), IEEE Computer Society.

73. Zave, P., and Jackson, M. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology 6* (1997), 1–30.

74. Zowghi, D., and Offen, R. A Logical Framework for Modeling and Reasoning about the Evolution of Requirements. In *Int. Conf. on Req. Engineering* (1997), pp. 247–257.