Software Evolution: a Requirements Engineering Approach

by

Neil Alexander Ernst

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Software Evolution: a Requirements Engineering Approach

Neil Alexander Ernst

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2012

This thesis examines the issue of software evolution from a Requirements Engineering perspective. This perspective is founded on the premise that software evolution is best managed with reference to the requirements of a given software system. In particular, I follow the Requirements Problem approach to software development: the problem of developing software can be characterized as finding a specification that satisfies user requirements, subject to domain constraints.

To enable this, I propose a shift from treating requirements as *artifacts* to treating requirements as design knowledge, embedded in *knowledge bases*. Most requirements today, when they exist in tangible form at all, are static objects. Such artifacts are quickly out of date and difficult to update. Instead, I propose that requirements be maintained in a knowledge base which supports knowledge-level operations for asserting new knowledge and updating existing knowledge. Consistency checks and entailment of new specifications is done automatically by answering simple queries. Maintaining a requirements knowledge base in parallel with running code means that changes precipitated by evolution are always addressed relative to the ultimate purpose of the system.

This thesis begins with empirical studies which establish the nature of the requirements evolution problem. I use an extended case study of payment cards to motivate the following discussion. I begin at an abstract level, by introducing a requirements engineering knowledge base (REKB) using a functional specification. Since it is functional, the specifics of the implementation are left open. I then describe one implementation, using a reason-maintenance system, and show how this implementation can a) solve static requirements problems; b) help stakeholders bring requirements and implementation following a change in the requirements problem; c) propose paraconsistent reasoning to support inconsistency tolerance in the REKB.

The end result of my work on the REKB is a tool and approach which can guide software developers and software maintainers in design and decision-making in the context of software evolution.

Ignoramus et ignorabimus ... "we do not know and will not know"

We must not believe those, who today, with philosophical bearing and deliberative tone, prophesy the fall of culture and accept the *ignorabimus.* For us there is no *ignorabimus*, and in my opinion none whatever in natural science. In opposition to the foolish *ignorabimus* our slogan shall be: **Wir müssen wissen – wir werden wissen!** ('We must know – we will know!') – *David Hilbert*

Acknowledgements

Many people have helped shaped this, the final result of my student years. This thesis may have my name on it but the majority of the work was done in the best traditions of scientific collaboration. I would like to thank my committee members Steve Easterbrook and Kelly Lyons, as well as erstwhile member Greg Jamieson, for excellent advice on scope and direction. My external examiner, Vincenzo Gervasi, made substantial improvement possible through his insightful comments.

In particular I wish to thank Jorge Aranda, Yiqiao Wang, Alexei Lapouchnian, Jennifer Horkoff and Rick Salay for always answering my questions and provoking yet more. The consistent high standards of our lab are an inspiration and a challenge. Thanks to Abram Hindle for being a truly collaborative research partner.

I was fortunate enough to spend three months visiting the University of Trento, and made great friends there, including Fabiano Dalpiaz, Amit Chopra, Raian Ali, Vitor Souza, Michele Vescovi and Alberto Griggio. I am indebted to them for welcoming me to the group, and making me feel at home.

To Ivan Jureta, your commitment to the issue of the requirements problem has made my own work much stronger.

For Alex Borgida, my co-supervisor, many thanks for your patience and interest. In explaining things to you I inevitably clarified my own thinking dramatically. Thank you for being understanding of a geographer turned part-time language designer. And naturally, many thanks to my co-supervisor and mentor, John Mylopoulos. Oftentimes I would pay too little heed to what you advised; it inevitably turned out to be the right approach. Any use of the personal pronoun herein should be seen as convention, because both Alex and John have greatly contributed to, helped with, and influenced all my work.

My family have always been there for me, unquestioningly supportive. Sometimes it is nice not to have to explain things.

To Kambria, "Love is not a big enough word. It's not a big enough word for how I feel about my wife."

For Elliott, who wasn't yet there in person but was great motivation to wrap things up, and for Kieran, for showing me that thesis writing can be fun if you occasionally pause to enjoy the simple things, like throwing a ball. This thesis is dedicated to you guys.

Contents

List of Figures

List of Tables

List of Algorithms

List of listings

Chapter 1

Introduction

1.1 Software Must Accommodate Unanticipated Change

Creating and maintaining software is a design activity [Jarke et al., 2010]. The outcome of the design activity is a software program[1] that satisfies a set of requirements. The design activity can create a new piece of software, or more commonly, re-design existing software to accommodate changes. There are three elements involved in the design activity. The first consists of the properties that are assumed to hold in the domain and are exploited in designing the software—the **domain assumptions** under which the software will operate. Next, there are the desired set of new properties of that domain—the **requirements** the software must meet for its users to accept it. Finally, there is the detailed plan for implementing the software—the **specification**. These three elements are related: if the software is developed according to the specification, taking into account the domain assumptions, the software system will satisfy the requirements [Zave and Jackson, 1997].

Challenges arise when these elements change. For example, the users might decide they have a new requirement that is vital to their continued acceptance and use of the software. The specification might need revision to accommodate new understanding of implementation costs. Finally, the assumptions made about the operating domain, such as the legal obligations the software bears for user privacy, might be invalidated by new legislation.

Designers and maintainers of the software cannot anticipate all possible changes. By definition, an anticipated change is one which was taken into account during design.

In other words, if a particular change was anticipated, then a specification would have been chosen

[1] Here the term 'software' is aggregating many concepts: software ecosystems, service computing, Software As A Service, a Machine, cloud computing, etc.

and a software system implemented that could accommodate that change.

Consider the case of a webserver's availability. In many modern web servers, accommodation for sudden spikes in server load (e.g., unexpected news stories such as a terrorist attack) is part of the software. The designers have anticipated the possibility of these sudden spikes. And yet servers are considered reliable if they can maintain "five 9s" for availability. That availability is not 100% tells one that there are always unanticipated (and costly!) events.

As with any endeavour, the future is always unpredictable. Consequently, the process of software design is inherently uncertain, whether for an entirely new system or for maintenance of an existing system. Consider the following quotation from Donald Rumsfeld:

> As we know, there are known knowns. There are things we know we know. We also know there are known unknowns. That is to say we know there are some things we do not know. But there are also unknown unknowns, the ones we don't know we don't know. *(Press briefing, Feb 2002).*

The statement "... there are also unknown unknowns, the ones we don't know we don't know" may seem like sophistry on its face, but does refer to a complex problem. Consider this statement in the case of building change-tolerant software for spacecraft. Our *known knowns* might include fundamental properties such as the gravitational constant. Our *known unknowns* are those properties we know at design-time we will need to monitor, such as the stochastic properties of complex, non-linear systems.

Our *unknown unknowns*, or unanticipated changes, are the most problematic. These are the possible situations we have not anticipated. NASA systems are well known for multiple redundancies, but one can only construct redundancies for situations that are anticipated. This was famously demonstrated in the Apollo 13 disaster. No one anticipated that the spacecraft for landing on the Moon would be used to host three astronauts on a return trip to Earth. The mission specification called for the lander to be left on the Moon. A chance meteoroid strike put paid to that assumption, however, and some clever engineering was required to adapt to the new situation. *Unknown unknowns* may either be truly unanticipated, or merely not anticipated *in the system*: planners know about the possibility, but are constrained in attempting to mitigate it, perhaps due to time or financial constraints.[2]

The problem of handling 'unknown unknowns' in software development is relevant to new projects, which are started from a blank slate. They are also relevant to the more common scenario of "brownfield

[2]There is a fourth possibility: *unknown knowns*. These are concepts we actually do understand, but do not remember or consciously believe. This is tacit knowledge, like riding a bicycle: easy to do, but not easy to transfer. In the context of changing systems, unknown knowns might include sociological phenomena like the potential problems between an organization which is based in the metric system, and one which uses Imperial units. In this case, it could be 'known' that this might be an issue, but for reasons of institutional inertia or bureaucracy is ignored.

projects", which are software projects with pre-existing, legacy artifacts, including some possibly implicit requirements. In this context, managing change involves understanding how to work with the existing implementation, and falls under the category of software maintenance.

Software maintenance consumes a large amount of the resources devoted to a company's information technology budget. Many researchers have shown this to be the case, including Boehm and Papaccio [1988] and Jones [2008]. Swanson [1976] and Lientz et al. [1978] showed that a large portion of maintenance was dedicated to the perfection of the software (51%) or to the adaptation of the software (24%). In Swanson's taxonomy perfective maintenance is carried out to accede to new user demands or to make the system work faster. Adaptive maintenance is undertaken to adjust to changes in operating conditions, which I call domain assumptions. Comparatively less effort (22%) was devoted to correction, which nonetheless consumes considerable attention in the software engineering literature.

Maintaining software is often associated with working with an implementation (and therefore the specification of that implementation). If the initial impetus for such a change was corrective, this is easy enough. For example, the change might correct a buffer overflow condition. In that case the requirements and domain assumption were correct, and the specification was shown to satisfy the requirements. However, the implementation deviated from the specification.

While corrective maintenance is important, this thesis focuses instead on the changes an engineer should carry out if the maintenance problem is perfective or adaptive. Perfective or adaptive changes must be undertaken with respect to the wider design problem. This design problem is captured by the interactions of the underlying requirements, domain assumptions and specification. The requirements define what ought to be, and the domain assumptions define the current state of the operating environment. If a maintenance activity, such as updating source code, does not refer to either requirements or domain, it raises the question, "Why is this change being made?" The central task when faced with an unanticipated change, when trying to move an event from an "unknown unknown" to a "known known", is to ensure the relationship between the three components still holds. This is done by testing whether the specification and assumptions about the domain still imply satisfaction of the requirements.

The three "Cs" of Consistency, Completeness and Correctness [Zowghi and Gervasi, 2003] are a useful perspective on the nature of the relationship between requirements, domain assumptions and specifications. Zowghi and Gervasi argue Requirements Engineering is a process which progresses iteratively from "rough sketches to correct requirements specifications"'. At each cycle the engineer must ensure that the requirements specification is correct. A Correct software system is one which is both Complete and Consistent; `Complete' means the system satisfies every requirement, and`Consistent' means the system has no internal contradictions.

This notion of RE as an iterative process of refinement persists beyond the initial, validated specification, however. Maintaining the software is also a process of iterative adjustment of the requirements specification. If the relationship between the domain assumptions, requirements, and specification is inconsistent, maintainers can no longer determine which requirements are satisfied, which domain assumptions are respected, or which specification has been implemented. Furthermore, efforts to improve, say, Consistency, are often at the expense of Completeness: consistency is easier to achieve if completeness of the requirements is relaxed. Similarly, attempting to make a set of requirements more complete can introduce inconsistency as new, conflicting requirements are introduced.

If the environment has changed, maintainers may need to make a corresponding correction in the specification or the requirements to ensure completeness. Thus, the relationship between these components is *guiding* maintenance activity. If a change results in an inconsistent system, then some other update must happen (such as adding a task to the specification, relaxing domain assumptions) to resolve the Inconsistency. There is no other rational way to select the appropriate change to make.

A related aspect of the relationship between specification, domain and requirements is optimality. Typically there will be multiple choices for ensuring Completeness of a software system. Given this, a further constraint is to select the optimal change to make. Consider a maintenance change which is carried out in an ad-hoc or intuitive manner, that is, without reference to the requirements specification. Such a change might well ensure Consistency, Completeness, and be Correct with respect to stakeholder needs. However, such a change may ignore the full range of options available, and result in a sub-optimal, if correct, system. This observation leads to the central message of my thesis: to systematically determine the optimal change to undertake, one must maintain models of all three components throughout the software's lifecycle.[3]

Failing to maintain the relationship between specification, domain and requirements is risky. It is true that costly software failures result from implementation mistakes, such as floating point errors in the Ariane-5 disaster [Nuseibeh, 1997], or unit mismatch in NASA's Mars Climate Orbiter. However, software failures are also due to a misunderstood relationship between all three components. [4] This is true of the untraceable requirements in Ariane-5 and the unjustified assumptions for the London Ambulance Service [Finkelstein and Dowell, 1996].

[3]In Chapter 4 I discuss some objections to insisting these models co-evolve.

[4]Arguably, all errors are symptomatic of a wider, systemic problem such as poor management or process.

1.2 Software Development Solves Requirements Problems

Software design involves an important distinction between *indicative* and *optative* descriptions. *Indicative* descriptions are "as-is" descriptions. They describe properties in the domain and model the state of affairs as it stands at the time of analysis. An indicative statement is "75% of customers have credit cards".

Optative descriptions are statements about desired, "to-be" properties of the domain, which are the requirements. For example, a soccer stadium may want the capability to accept credit card purchases for souvenirs: "Accept credit cards for payment". The challenge for software designers is to relate the indicative descriptions to the optative descriptions. Zave and Jackson [1997] defined this formally as the **requirements problem**: find correct specifications S, which in conjunction with indicative domain knowledge W, imply the satisfaction of the optative requirements R. This implies that a solution to this problem will be some specification S. Therefore answering the requirements problem means a) searching for a specification S b) correctly modeling the constraints imposed by the domain W, and c) understanding the requirements R we want to satisfy. A solution to the requirements problem will be Correct, Consistent and Complete, as described in [Zowghi and Gervasi, 2003].

If specification means "what is to be built", the term "requirements specification" is misleading: the requirements representation does not say anything about the form of the solution. The representation merely guides implementation decisions, captured in the specification, such that the specification will meet the requirements. The nature of a specification S is deliberately vague in this thesis. Other research has looked at how to translate from requirements to implementation. In my work, the notion of specification will refer to a generic implementation. This implementation might consist of services, components, tests, agent responsibility assignment, or a formal specification. In this thesis, I represent these potential members of the specification as tasks that must be performed in some order, which is left undefined.

Since we are concerned with changes to a given requirements problem, it is important to note that what was indicative at one point might change at another: we might learn that "All customers have credit cards". And what was optative will be indicative, once the software has been created: "Stamford Bridge accepts all major credit cards". This implies that our relation between W,S and R will be non-monotonic: some of the information in W, S, or R might change from version to version. An indicative description might include the fact that certain portions of S have already been implemented, if we are doing "brownfield" development.

The Requirements Evolution Problem – Suppose now that we have acquired and solved a requirements problem RP1. "Solved" here means we have a running solution, consisting a set of tasks in S, which have been correctly implemented. As I argued earlier, we are likely to encounter unknown unknowns, the unanticipated changes to the problem, which include changes to all aspects of RP1, including the requirements R, tasks in S, and domain knowledge W. The effect of this change is a new requirements problem RP2. Formally, the obvious path to follow in this case would be to view RP2 as a completely new problem, and simply search for a new solution. However, this view is unrealistic in the real wold. For example, one does not throw away an entire software system and start design from scratch when a new feature is desired. Instead, the solution is likely to be *incremental*. We start from a current solution and try to move towards one that meets the problem captured in RP2. The *requirements evolution problem* is the search for a solution to a revised requirements problem such that the new solution is in some way related to the original solution. The differences between solutions is captured in terms of the suggested sets of implementation tasks for each solution.

1.3 Research Questions and Contributions

This section links my specific research contributions with the research questions those contributions answer. The Requirements Engineering perspective on software evolution is captured in the following research question: *How does one find algorithmically optimal solutions to the requirements problem following unanticipated changes?*

Let me examine the parts of this question in more detail. The question refers to "finding solutions" because, as we have seen, the requirements problem delineates a space of possible solutions, for which we must choose one. The term "unanticipated" constrains my thesis to software evolution that involves "unknown unknowns". The challenge of designing adaptive software (I discuss the difference in detail in Chapter 8) is put aside. The term "optimal" emphasizes the importance of finding optimal solutions, with respect to user desires. Finally, as we have seen, the requirements problem is central to my research. My thesis is centred around a system for solving requirements problems. Following the criteria given by [Zowghi and Offen, 1997], such a system must have the following properties:

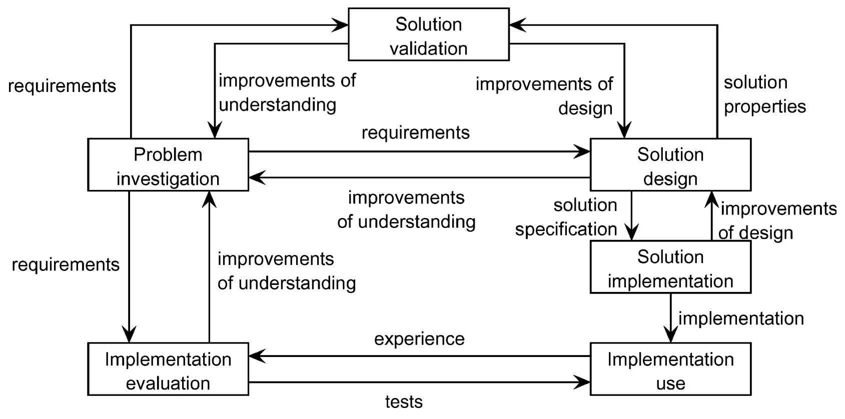
1. Support reasoning even in the presence of inconsistent problem formulations.

2. Find minimal solutions to the requirements problem for a given set of mandatory requirements.

3. Identify new solutions when the problem formulation changes.

4. Support a partial order over goals that determine which goals to prefer when revising. Such a partial order may be defined by any number of criteria, including cost, age, and change-proneness.

Figure 1.1: Wieringa et al. [2006]'s diagram showing activities in the engineering cycle. Boxes represent activities, arrows represent impacts. There is no preferred sequential relationship among the activities.



The thesis will show how the system I created achieves these properties. To outline the steps I have taken, I will outline my approach using Wieringa et al. [2006]'s research framework, which they call the engineering cycle, as illustrated in Figure 1.1. This framework applies the well understood notion of the engineering cycle to requirements engineering. The engineering cycle may start at any point, but typically contains *problem investigation*, *solution design*, *solution validation*, *solution selection*, *solution implementation*, and *implementation investigation.* Note that this use of 'implementation' is referring to implementing a solution in practice. Each phase of the engineering cycle answers a particular research question. Questions at the problem investigation stage will be *exploratory*: "why is this happening?" "What is the nature of the phenomenon?" Questions at the solution design stage are *confirmatory*: "Does this solution solve the problem?" *Validation research* questions will seek to combine the new design with the initial understanding of the problem: "What does the problem now look like? What changes does my new design bring about?"

The thesis makes three types of contributions. The first type consists of concepts for providing a better understanding of requirements problems. The second type is a methodology for solving such problems. The third type are tools, physical artifacts that help to resolve the problems. I approach my research question in stages; at each stage I conclude with a discussion of validation and related work. Chapter 2 describes some important preliminaries, while Chapter 3 gives a broad overview of research in the area. *Problem investigation* occurs when I investigate the characteristics of the requirements evolution problem in Chapter 4, expanded with a detailed case study in Chapter 5. I propose *solution designs* in Chapter 6. In Chapter 7 I design a solution and validate its utility, using what is called *validation research* in Wieringa et al. [2006]. I consider other problem characteristics in Chapters 8 and 9. I summarize my findings and discuss future work in Chapter 10.

The following tables provide a more detailed look at each of the core chapters, with respect to Wieringa et al. [2006]'s research framework.

Chapters 4 and 5 investigate the problem of "unknown unknowns" in software evolution. In Chapter 4 I conduct several empirical studies of changing requirements, using machine learning tools. I show that requirements clearly are changing, and that this change is often not anticipated. Chapter 5 introduces a large-scale study of the requirements evolution problem in the payment card industry. I use this example throughout the thesis.

Chapter	**4 and 5**
Research question(s)	Is there really a problem with unanticipated changes? How are requirements changing?
Engineering phase	Problem investigation.
Concepts introduced	Wordlists derived from cross-project quality taxonomies.
Methods for	Comparing quality requirements between projects, and over time; Comparing empirical studies of evolving requirements.
Tools developed	Machine learning tools for extracting quality requirements.
Publications	Ernst and Mylopoulos [2007, 2010]; Hindle et al. [2011]

Chapter 6 leverages the empirical results to inform the design of a specification for a system that solves requirements problems. I call this system the REKB. I consider how the artifacts in the requirements problem, that is, the requirements, the specification, and the domain assumptions, ought to be managed. The goal of the REKB is to provide a black box for finding solutions to the requirements problem.

Chapter	**6**
Research question(s)	How should these artifacts be managed? How will we interact with them, (e.g., add new requirements, remove old requirements, update the domain assumptions) and use the artifacts for reasoning tasks?
Engineering phase	Solution design.
Concepts introduced	The Requirements Engineering Knowledge Base (rekb); specification-level operations for working with rekbs; optimal solutions in the context of unanticipated change; minimal solutions to requirements problems.
Methods for	Enabling users to solve requirements problems; deciding which solution to choose, based on the degree to which a solution reuses elements of the previous solution.
Tools developed	*not applicable.*
Publications	Ernst et al. [2011]

Having defined a functional specification, I turn to implementing this specification such that actual requirements problems can be considered. Chapter 7 considers such an implementation using an assumption-based truth maintenance system. The chapter then considers the complexity of the reasoning task for finding minimal solutions. Finally, this chapter examines an approach to finding optimal solutions.

Chapter	**7**
Research question(s)	Can we support the operations defined on the REKB efficiently? At what size of problem will the implementation fail?
Engineering phase	Solution implementation.
Concepts introduced	Reason-maintenance systems; abductive reasoning with requirements models.
Methods for	Creating random requirements models for experimentation; Assessing the practicality of a REKB implementation.
Tools developed	REKB tool supporting operations in Chapter 3; domain-specific language for interacting with REKB; requirements model generator using random graphs. Algorithms for local search of requirements.
Publications	Ernst et al. [2008]; Jureta et al. [2010]; Ernst et al. [2010, 2011]

Where Chapter 7 considered only static, consistent requirements problems, Chapter 8 will focus on the "unknown unknowns", showing how they can be accommodated in the REKB. This is the problem of revising requirements problems. Again, the problem is assumed to be consistent.

Chapter	**8**
Research question(s)	What are the rules for changing a REKB after a revision? What should we do if our requirements are conflicting? Which requirements are important to keep?
Engineering phase	Solution design (what operations are necessary to do this for an REKB).
Concepts introduced	Version control of requirements; Requirements revision.
Methods for	Choosing a solution in spite of contradictory evidence.
Tools developed	Extensions to reason-maintenance to revise requirements.
Publications	Ernst et al. [2008], Ernst et al. [2011]

The final core piece of my thesis will turn, at long last, to the issue of inconsistent requirements problems. Inconsistency occurs when there is a conflict in the requirements problem, such that the requirements cannot be satisfied. Chapter 9 will focus on the difficulties this poses for the REKB approach. In particular, I consider what principles a paraconsistent REKB ought to respect.

Chapter	**9**
Research question(s)	What new considerations are introduced when we allow the REKB to be inconsistent?
Engineering phase	Problem investigation (how current tools handle inconsistency);
Concepts introduced	Paraconsistent reasoning with requirements; minimally inconsistent subsets of requirements;
Methods for	Reasoning with inconsistent requirements problems;
Tools developed	Operators for inconsistent REKB.
Publications	Jureta et al. [2011]

1.4 Publications

In order to clarify the extent of my scholarly contributions, this section outlines the scope of my work with respect to academic publications. The best academic work is collaborative, but for a thesis, it is important to clarify the extent of one's intellectual contributions. The following are publications for which I was a co-author and are relevant to this thesis. As I mention in the acknowledgements, when I say *me* or *I* in this thesis, I am including the close work I did with my supervisors Alex Borgida and John Mylopoulos.

For Chapter 4 the following work is important: Ernst and Mylopoulos [2007, 2010]; Hindle et al. [2011]. The first paper was entirely my own work with advice from a number of other parties on how to shape the paper, particularly from historians on how a software history might look. I then took this approach and expanded it with the data-mining effort on open source software, published in the second paper. Again, this was entirely my own work, but in order to strengthen the analysis, I began consulting with Abram Hindle, now assistant professor at the University of Alberta. This turned into a collaborative partnership that resulted in our publication [Hindle et al., 2011]. Abram was responsible for data selection, preparation and statistical analysis; I originated the notion of using software qualities to label topics; we both annotated the data for supervised classification and wrote up the results.

When I introduce what I call the REKB (Requirements Engineering Knowledge Base), I leverage work by Ivan Jureta on Techne, starting with Jureta et al. [2008], a paper with which I was not involved. Following some meetings on how we might expand Ivan's CORE ontology with tool support and semantics, we wrote the paper Jureta et al. [2010], which defined a syntax and semantics for what we called the Techne requirements modeling language. Ivan Jureta and Alex Borgida were primary technical contributors; I contributed an initial implementation of the language which showed the flaws in the original logical formulation, and helped to debug the logic. From that, I wrote Ernst et al. [2011], which expands on Techne for the case where the requirements problem is evolving, and also introduces the notion of a functional description for requirements problems. I and Alex Borgida were primary technical contributors, and build on our earlier work.

When this thesis talks about implementation, particularly the algorithms in Chapter 7, that is entirely my contribution, with help from Alex Borgida in tightening logical aspects of the problem. I build on the work of [de Kleer, 1986] on the Assumption-Based Truth Maintenance System ATMS), which is a reasoner (in the same sense that the zChaff SAT solver is for [Sebastiani et al., 2004]) that I use for solving requirements problems. I worked closely with Yijun Yu on the implementation and algorithms for model versioning, which we published as a poster in Ernst et al. [2008]. This work leveraged the infrastructure of Molhado, developed by Tien Nguyen.

Finally, the Tabu search approach described in Chapter 8 is my application of an existing search strategy to that particular problem, which I described in Ernst et al. [2010].

Any mistakes in this thesis are of course my own.

Chapter 2

Preliminaries

Three areas of study are important to cover in detail in order to properly understand the remainder of the thesis. Requirements Engineering (RE) is the commonly-applied term for solving requirements problems. As with all engineering disciplines, RE is "codifying scientific knowledge about a technological problem domain in a form that is directly useful to the practitioner, thereby providing answers for questions that commonly occur in practice ... engineering shares prior solutions rather than relying on virtuoso problem solving [Shaw, 1990]." Goal-oriented requirements engineering (GORE, discussed in §2.1) is the requirements engineering modeling perspective I use in the thesis. GORE can be distinguished from approaches which focus on social aspects of software design or approaches which are interested in the functional description of the system-to-be's behaviour.

In §2.2 I briefly introduce the concepts from propositional logic that I use in my thesis. The final section (§2.3) looks at knowledge level modeling, and why this is a useful lens for understanding how a system ought to operate.

2.1 Goal Modeling in Requirements Engineering

This thesis uses a goal-oriented approach to requirements engineering, so it is useful to explain what that is in some detail. Requirements are the desired properties of a new system which the specification must bring about. They are objectives the stakeholder wants fulfilled in order for him or her to accept the system. This naturally lends itself to describing the requirements as a series of goals for the specification to fulfill. Goals are desired states of affairs the new system will achieve, such as "support the use of VISA cards to pay for purchases".

Early requirements languages [Ross and Schoman, 1977; Bubenko, 1980; Greenspan et al., 1982;

Mylopoulos et al., 1990] focused on functional requirements, and answered the question "what do we need to build" (the major contribution being the recognition that "how we build it" is left to specification and design phases). It was soon recognized [Robinson, 1989; Chung et al., 1992; Dardenne et al., 1993; Antón, 1996] that a higher level of abstraction was necessary, one which would answer the question "why are we building this". Such an abstraction would capture the objectives of the system-to-be as told by the client. In particular, requirements researchers leveraged some of the work on goals from AI, where they were (and are) used to capture states of the world for problem solving and planning (starting primarily with Fikes and Nilsson [1971]'s STRIPS planner). A particularly useful approach is to *refine* high-level goals into constituent lower-level goals which can bring them about. For example, if I want to achieve the goal "support the use of VISA cards to pay for purchases", I could say that achieving this requires satisfying the goals "purchase credit-card terminals" and "open merchant account with VISA". This is known as AND-OR refinement, where a higher-level goal may have two or more sub-goals linked to it. These sub-goals can be linked disjunctively (OR) or conjunctively (AND). The presence of disjunction in the model represents design variability, since more than one alternative solution can be chosen. The resulting model may be easily converted into a formal logical representation as a propositional theory, which I discuss in the next section.

The other elements of a goal model, as I use the concept, are tasks, which are related to goals using a means-ends relation (i.e., performing the task implies satisfying the goal) and domain assumptions, which are (in a particular model instance) invariants from the domain. A domain assumption can express the relation between two goals ("I assume that satisfying this goal means I also satisfy this other goal"), or capture constraints on the model ("The server will receive as many as 70 simultaneous connections").

2.1.1 Turning Stakeholder Desires Into Goals

Jureta et al. [2008] provide a nice support for mapping between stakeholder communications and goal models by using Searle's speech act theory [Searle, 1976], as part of their CORE ontology. Speech acts are a way of analyzing human communications. According to Searle, when something is said, it consists of both the actual *content* as well as an *illocutionary act*. An illocutionary act characterizes the consequences of what someone just said. They are divided into

- representatives (the thing I said is the case), also known as assertives,
- directives (I would like you to do the thing I said),
- commissives (I will do this thing I said in the future),

- expressives (this is how I feel about this thing I said), and
- declarations (this thing I said changes the state of the world).

Jureta et al. mapped *directive* speech acts into goals: from our earlier example, if a client tells the developer to support the use of credit cards, he or she is expressing a desire that they will build a system which will satisfy this *optative* condition. *Commissives* are mapped into tasks, representing the commitment to carry out that task. *Assertives, representatives* and *declaratives* become beliefs about the *indicative* state of the world, which are domain assumptions under the ontology. Finally, expressives are treated as attitudes towards elements in the model. These attitudes are used to select among several candidate solutions to the requirements problem (e.g, as preferences between requirements). The act of eliciting and modelling requirements, then, is concerned with soliciting client/stakeholder speech acts, and then classifying those acts using the CORE ontology.

2.1.2 The Requirements Problem in **Techne**

I will introduce a formalization of the relationship between these speech acts: requirements (as goals), specification and domain assumptions. Formalizing these components is useful for three reasons:

- It clarifies the meaning of each of these components, as well as the relationship between them. A good formalization has only one meaning, which forces us to crystallize the translation from natural language to a particular formalization.
- It supports automated reasoning over the formal model. This is useful in detecting logical inconsistency, or to derive the necessary form of the specification.
- It disambiguates hitherto implicit relations and definitions.

Some object to the idea of formalizing requirements engineering, implying that requirements activities are necessarily rough and informal. There is an objection that formality poses an additional barrier to working with technically-unskilled stakeholders. Another barrier is that the rapid and incremental development called for by agile methodologies does not readily favor formal analysis. As we will see, however, the formalization in this thesis is quite simple: natural language statements of requirements remain as they are, and are clear to any stakeholder: what is formalized are some specific relations between these statements, and it is by looking at these relations that automated reasoning is performed (rather than by looking “into the content” of natural language statements) [Ernst et al., 2011].

The formalization I use will follow from a more precise definition of the relationship between the three components of specification, requirements, and domain assumptions. This formalization will define the **requirements problem**.

Zave and Jackson formalized this as the challenge of finding S that satisfies the condition:

$$W, S \vdash R \tag{2.1}$$

where $\vdash$ is logical consequence relationship. A further constraint is that $W \cup S$ is consistent. If the union of these two were not consistent, then our requirements problem would be trivially satisfiable, as I discuss in the subsequent section (§2.2).

Subsequent research (e.g., Chung et al. [1992]; Letier and van Lamsweerde [2004]; Castro et al. [2002]) has shown this characterization to be too abstract. First, given W and R, there can be *many* solutions S to the requirements problem. Clearly, not all members of S are equally desirable. Therefore, provisions should be made for exploring the space of solutions and comparing them. In my thesis, this will be a "workbench" metaphor based on a database. Second, requirements are not all alike, and include, among others, ones that are *mandatory* ("must-have") and others which are *preferred*, those the stakeholders consider "nice to have" *ceteris paribus*. Other requirements emerge during analysis and serve to structure the problem space using refinements. Finally, this characterization does not talk about the work required to expand W to the point that the proof works: i.e., add enough knowledge to W, e.g., in the form of requirements dependencies, to explain why the specification meets the requirements.

To address these deficiencies, we can once more turn to the CORE ontology introduced in [Jureta et al., 2008, 2009b]. CORE partly consists of three components, matching the three components of the requirements problem. The first aspect are *goals* G of various kinds and attitudes to them (*mandatory, preferred*). The second aspect are sets of *tasks* S referring to behaviours of the system-to-be. The final type considered in my thesis are *domain assumptions* D, which are conditions believed to hold in the world, as well as statements that describe how goals are refined or operationalized. This ontology was used as a basis for a new requirements modeling language, Techne [Jureta et al., 2010].[1]

In Techne, the requirements problem identified in Zave and Jackson [1997] is re-stated as the search for tasks S and refinements/realizations/constraints that can be added to the world knowledge, captured as domain assumptions D, such that goals are satisfied, i.e.,

$$D, S \vdash G \tag{2.2}$$

[1]In this thesis I use a simplified version of Techne, ignoring some variants of goals (e.g. soft vs hard) as well as "quality constraints" because they do not affect the fundamental notion of identifying solutions to the requirements problem.

Requirements problems are often structured somehow, representing abstractions from high-level requirements ("sell more product") to low-level tasks ("use Moneris payment terminals"). Therefore a key part of solving requirements problems is finding ways to refine requirements using other requirements, to further refine some requirements into tasks, and to record conflicts between requirements, giving rise to refinement (I) and conflict (C) relations. The existence of individual relations is considered to be part of our domain assumptions. That is, the existence of a refinement between two goals is an assumption that such a relation exists in reality (and is invariant).

Techne identified two stages in solving requirements problems. The first step is to identify all *candidate solutions* to the problem, that is, all sets S which satisfy mandatory goals.[2] The second step is to select from among those candidates the solution which is optimal, according to some definition of optimal and some decision rules. The only constraint is that a solution S which is dominated on all criteria by another solution *S'* is not considered. To this end, Techne introduced two decision criteria. One is the notion of preferred goals, those which are nice to have but not mandatory. The second is a binary relation between goals called a preference relation.

Let us consider a simple example demonstrating how a requirements problem is represented in Techne.

Example 1. *Our objective is to build a software system for a football stadium (e.g., Chelsea FC's Stamford Bridge). Our elicitation with the stakeholders has produced the following set of communications, sorted into tasks T, goals G, and domain assumptions D:*

- G_{IncRev}*: Increase revenue.*
- $G_{IncSales}$*: Increase sales.*
- T_{CC}*: Accept payment cards.*
- T_{Cash}*: Accept cash.*
- G_{NoLoss}*: Avoid financial losses and penalties.*
- G_{PCI_compl}*: Be PCI-DSS compliant.*
- D_NoRes*: The stadium management do not have resources to purchase multiple servers.*
- T_{Virt}*: Ensure virtualization instances are isolated from one another.*
- T_{Mult}*: Use multiple servers to isolate functions.*
- G_{FnServ}*: Implement only one primary function per server.*

We must add our relations in order to structure the problem. In our example, these relations[3] *look like:*

- I_1*: Satisfying* G_{NoLoss} *and* $G_{IncSales}$ *satisfies* G_{IncRev}*.*

[2]It is important to note that the space of possible solutions S is limited by the tasks present in the model of the requirements problem, and is necessarily incomplete (i.e., missing those tasks which have not yet been elicited). Furthermore, it is trivially easy to satisfy the requirements problem by including "no-op" tasks such as "this task satisfies the goal". We assume that this problem is independent of the manner in which possible solutions are found.

[3]Properly the labels should be prefaced with D to reflect their status as domain assumptions in Techne. See §9.1 for a longer discussion of this point.

- *I_2: Satisfying G_{PCI_compl} satisfies G_{NoLoss}.*
- *I_3: Satisfying G_{FnServ} (partially) satisfies G_{PCI_Compl}.*
- *I_4: Doing T_{CC} satisfies $G_{IncSales}$.*
- *I_5: Doing T_{Cash} achieves $G_{IncSales}$.*
- *I_6: Doing T_{Virt} achieves G_{FnServ}.*
- *I_7: Doing T_{Mult} achieves G_{FnServ}.*
- *C_1: T_{Mult} and D_{NoRes} are in conflict.*

■

The natural language equivalent to, e.g., I_1: Satisfying G_{NoLoss} and $G_{IncSales}$ satisfies G_{IncRev} might be a statement "If we avoid financial losses and penalties, and also increase sales, then we will increase revenue."

Fig. 2.1 shows a graphical representation of the previous example.

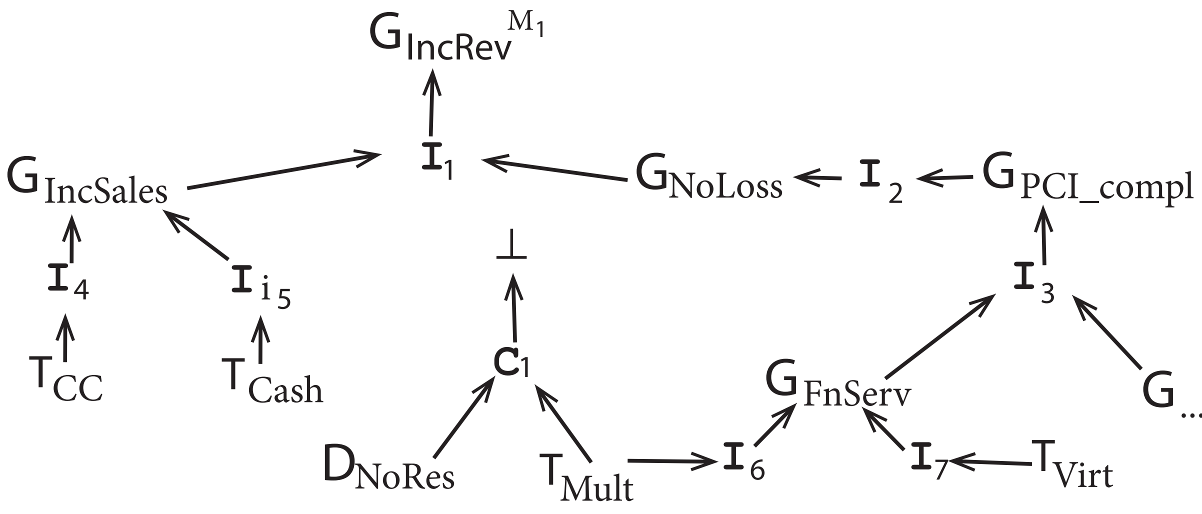


Figure 2.1: A simple requirements problem showing goals, tasks, domain assumptions, structured using refinement and conflict. Arrows indicate untyped relations.

The following example illustrates the second phase of solving a requirements problem. Here we search for one of many possible solutions.

Example 2. *What sets of tasks, if implemented, will satisfy the requirement G_{IncRev}? There are two refinements which lead to the satisfaction of $G_{IncSales}$, and so the acceptable solutions S to this requirements problem are precisely those tasks which satisfy one or both of the alternative refinements. Therefore $S_1 : \{T_{Virt}, T_{Cash}\}$, $S_2 : \{T_{Virt}, T_{CC}\}$ and $S_3 : \{T_{Virt}, T_C C, T_{Cash}\}$. In this thesis we focus primarily on solutions which are minimal with respect to the subset relation, and therefore we will tend to disregard S_3 as subsumed by S_1 and S_2.*[4] ■

[4]Furthermore, this thesis does not consider numeric valuations of tasks (e.g., cost).

Let us expand on the possible modeling problems this representation can capture. Designers can model, in a domain assumption, the idea that a company cannot afford to buy more servers. They can also represent that there is a conflict between this assumption and the task of using multiple servers to host separate servers. This conflict influences the possible choices for a specification. One such restriction is the need to design a workaround to this conflict, in this case by using virtualization (T_{Virt}). Notice that T_{Mult} does not appear as a possible solution. It does not make sense to have extraneous steps/tasks in such sets T, so, although there may be many solutions, one expects them all to be minimal.

To deal with unanticipated requirements, we need to view the requirements problem in the context of change over time. There are three places where evolution can occur: in the domain (D), in the specification (S), or in the requirements (G). The Requirements Evolution Problem, as stated in §1.2, can be semi-formalized as:

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

Properties Π will be functions with the new possible solutions and the original solution as input, and returning a set of possible choices with respect to the chosen property. These criteria are discussed in §8.3.1.

A note about notation. As the thesis progresses, I shall be moving from the natural language examples (like those above), into a more formal syntactic notation designed to model the specification, and finally describing implementation. At each 'level' I refer to the same concept in the world, but the meaning of that concept will change depending on the syntax and semantics (if any) of the representation.

Consider the following:

1. $\parallel A, B, A$ refines $C \parallel$ — The natural language contents of speech acts. For example, A might represent "pick up clothes", B "remove dishes" and C "clean room", and the refinement relation captures the statement "I believe that if I pick up my clothes my room will be clean".

2. $\{\mathbf{t}(\alpha), \mathbf{t}(\beta), \mathbf{k}(\mathbf{t}(\alpha) \rightsquigarrow \mathbf{g}(\gamma))\}$ — The speech act labelled with its illocutionary force, according to the CORE ontology of Jureta et al. [2008], and represented in a propositional language $\mathcal{L}$. $\mathbf{t}()$ and $\mathbf{g}()$ represent commissive and directive speech acts; the $\mathbf{k}()$ relation an assertive or declarative act capturing the belief that the refinement (represented by $\rightsquigarrow$) holds.

3. $[\![\alpha, \beta, \alpha \rightarrow \gamma]\!]$ — The contents of $\mathcal{L}$ mapped into an implementation, in this case an assumption-based truth maintenance system. α, β are nodes, with the assumption property. They map to the level above using the datum relation. γ is a plain node, and $\alpha \rightarrow \gamma$ is a justification.

The specifics of each level are not important at this stage; what is necessary is to understand that while the various representations appear similar syntactically, they have different semantics.

2.2 Formal Background

Some background in propositional logic is useful for understanding later parts of the thesis. Parts of this section are derived from [Reeves and Clarke, 2003; Russell and Norvig, 1995].

We start from a finite set of propositions. A proposition is a statement (fact) which can be either true or false. A customer goal "Accept credit cards" is a proposition. Note that it is not relevant (at this stage) *how* we determine the truth of the proposition. This is even the case with softgoals, whose satisfaction is not testable algorithmically. A softgoal (such as "the system shall be secure") is nonetheless a proposition in our world, since it can only be true or false. This is in contrast to a probabilistic approach where a proposition is more or less likely to be true, or a fuzzy logic approach where a proposition could be more or less true.

2.2.1 Syntax of Propositional Logic

An atom is an individual proposition, which can be represented by a propositional variable (alt. symbol, letter): A stands for "Accept credit cards". A literal is a propositional variable A or its negation (represented $\neg A$). We use the special symbol $\perp$ to represent falsity and $\top$ to represent truth in the language. We connect propositional variables with three Boolean connectives: $\rightarrow, \vee, \wedge$, which mean material implication, disjunction, and conjunction, respectively. Combining literals and connectives according to the BNF grammar defined below (taken from Russell and Norvig [1995]) produces well-formed formulas (wffs, alt. sentences).

$\langle \textit{Sentence} \rangle ::= \langle \textit{AtomicSentence} \rangle \mid \langle \textit{ComplexSentence} \rangle$

$\langle \textit{AtomicSentence} \rangle ::= \top \mid \perp \mid \langle \textit{PropositionalVariable} \rangle$

$\langle \textit{ComplexSentence} \rangle ::= (\ \langle \textit{Sentence} \rangle\ ) \mid \langle \textit{Sentence} \rangle\ \langle \textit{Connective} \rangle\ \langle \textit{Sentence} \rangle \mid \neg\ \langle \textit{Sentence} \rangle$

$\langle \textit{Connective} \rangle ::= \wedge \mid \vee \mid \rightarrow$

$\langle \textit{PropositionalVariable} \rangle ::= \mathrm{P} \mid \mathrm{Q} \mid \mathrm{R} \mid ...$

A *clause* is a disjunction of literals, e.g., $(P \vee Q)$. We sometimes represent clauses as sets of literals $(\{P, Q\})$. A standard way to represent formulas is in Conjunctive Normal Form (CNF), conjunctions of clauses. The sentence $(P \vee Q) \wedge (R \vee S)$ is in CNF. We can represent CNF formulas as sets of clauses, so the previous example can also be represented as $S = \{\{P, Q\}, \{R, S\}\}$. A set of sentences is called a *theory*; an example of a theory is the REKB from Chapter 1.

A *Horn clause* is a clause with at most one positive literal. $\neg P \vee \neg Q \vee R$ is a Horn clause. Since the Boolean definition of implication is that $\neg A \vee B$ is equivalent to $A \rightarrow B$, we can translate any Horn clause in CNF to a corresponding implication: our previous formula becomes $P \wedge Q \rightarrow R$. Horn clauses with no positive literal, such as $\neg P \vee \neg Q \vee \neg R$, will be represented as $P \wedge Q \wedge R \rightarrow \perp$.

Propositional sentences derive meaning from their components. Truth tables can be used to understand the overall meaning. Table 2.1 shows one for implication. Note the last row, which states that if P is False $(\perp)$ we can conclude whatever we wish.

2.2.2 Rules of Inference

A language's rules of inference allow one to manipulate the formulas to derive new formulas symbolically. For our purposes, the important rule of inference is *modus ponens*: from an implication and an antecedent, infer the consequent. Proof rules are represented using horizontal lines, to show that the formulas on the top can be used to derive those on the bottom. For *modus ponens*, we represent it as:

$$\frac{P \rightarrow Q, P}{Q}$$

Another rule of inference of classical logic is the principle of *ex falso quodlibet*: from falsity derive everything [Hunter, 1998]. We write this formally as

$$\frac{P, \neg P}{Q}$$

We say that this *trivializes* the language, because Q can be any and all syntactically valid sentences. Note that P and Q must either be asserted or derivable from other sentences in the theory. This is exactly the case where the domain assumptions W are inconsistent with the implementation tasks S: all requirements are trivially satisfiable.

2.2.3 Semantics of Propositional Logic

An important property of a logic is to be able to assign meaning to a formula; meaning for us will mean a truth-value i.e., True,False. A *valuation* is an assignment of truth-values (True,False) to propositional variables in the language. This is is then extended recursively to complex formulas. The rules for such

Table 2.1: Truth table for implication

P	Q	$P \rightarrow Q$
$True$	$False$	$True$
$True$	$True$	$True$
$False$	$False$	$False$
$False$	$True$	$True$

Table 2.2: Truth table for G

S	$\neg S$
$True$	$False$
$False$	$True$

recursive valuations can be captured in truth tables; a valuation corresponds to one row of the truth table.

Table 2.1 shows the truth table for implication. Note the last row, which states that from a false premise one can conclude anything.

Formally, a valuation is a function $v : P \rightarrow \{\mathsf{True}, \mathsf{False}\}$ where P is a set of propositional variables. If a formula is valid for all possible valuations $v \in \mathcal{V}$, it is a tautology. For a set of formulas $G = \{S_1, \ldots, S_n\}$, if there is a valuation v that makes all $S_i \in G$ true, then G is consistent; if there is no such valuation, then G is inconsistent. Consider Table 2.2. G consists of $\{S, \neg S\}$; G is inconsistent, because there is no row in the table in which both sentences are true.

A set of formulas G *semantically entails* a formula T, written $G \vDash T$, if there is no valuation that makes all sentences in G true and also makes T false.

Model theory is the branch of logic which is concerned with the interpretation of a language (here propositional logic) using set-theoretic structures which define worlds in which a given sentence of the language is true or false. $Mod(T)$ denotes the set of all models of T (propositional valuations v satisfying T).

2.2.4 Abduction

Logical deduction is to reason from facts and rules to derive new conclusions (e.g., "If it rains the ground is wet", and "It is raining", means we can conclude "The ground is wet"). Logical *abduction* means to reason from conclusions (observations like "the ground is wet") and rules/theories ("If it rains the ground is wet") to abduce an explanation ("It is raining"). Since there are possibly many explanations for an observation ("The sprinkler is on", for example), we must filter our possible explanations, and define how we will select the most plausible observations.

Given a theory Σ and an observation s, a formula α is an explanation for s iff $\Sigma \cup \alpha \models s$ and $\Sigma \cup \alpha$ is consistent. α is a minimal abductive explanation if there is no "weaker" α' which also explains s (i.e., $\{\alpha\} \models \alpha'$. For example (from Selman and Levesque [1990]): if $\Sigma = \{\{p\}, \{q\}, \{\neg p, \neg r, \neg s, t\}\}$ then $r \wedge s$ and t are explanations for t, where t is the trivial explanation.

2.2.5 Propositional Satisfiability

The final concept from propositional logic which is relevant here is that of propositional satisfiability (SAT). Given a propositional theory Σ, we say Σ is satisfiable iff there is a valuation v to the propositional variables in Σ which makes every formula in Σ evaluate to true. Consider the theory $\Sigma = \{(P \vee Q) \wedge R\}$. Then Σ is satisfiable because the assignment $\{P \mapsto True, Q \mapsto False, R \mapsto True\}$ makes Σ True as well. Many important problems can be represented as satisfiability problems, and there are excellent tools available to search for satisfying valuations v for a given theory Σ, known as satisfiability solvers. Cook [1971] showed that this problem was NP-hard, and therefore answers are easy to verify but very difficult to find (in the worst case). For the theories in this thesis, however, SAT answers can be found in linear time, since Horn theories are linear-time satisfiable [Dowling and Gallier, 1984]. The abductive inference problem, covered above, is however NP-hard, in that it can be simulated, for arbitrary propositional formula, as an NP problem with an NP oracle [Eiter and Gottlob, 1995]. For the Horn clauses here considered, the problem is 'merely' an NP problem with a polynomial oracle, i.e., one level lower in the polynomial hierarchy.

2.2.6 Belief Revision

Belief revision is the study of the correct action to take when a representation of knowledge changes. The seminal paper on the subject is Alchourrón et al. [1985], with a more approachable description contained in Gärdenfors [1992]. These definitions assume a consistent set of propositions Δ and a consistent (classical) logical entailment relation $\vdash$. Recall that Δ is a set of propositions representing the requirements problem. There are three principle categories of changes:

1. **Belief expansion**: Add a consistent formula A to Δ such that the new Δ^+ is the set $\{B \mid \Delta \cup \{A\} \vdash B\}$. That is, the logical consequence of Δ^+ together with A. The new Δ^+ is consistent when A is consistent with Δ.

2. **Belief contraction**: A formula in Δ is removed. In order for the logical consequences of the remaining formulas to be closed, some other formulas must be abandoned. Since there is no logical way of determining which ones to abandon, an extra-logical choice function must be defined which will produce the new set Δ^-.

3. **Belief revision**: A formula not in Δ is added, but is not consistent with the other formulas in Δ. Therefore, similarly to retraction, an extra-logical choice function must be used to decide which formulas to give up to produce the new set Δ^*.

To see why this function is necessary, consider the simple case where $\Delta = \{A, B, A \wedge B \to C\}$ and its consequences, one of which is C. If we 'discover' a new formula, $\neg C$, then we must revise Δ with the new information (which is inconsistent with Δ). Clearly, we must abandon C as a consequence, but to do so implies removing one of the other formulas that produces C as a consequence (so that the new Δ^* is consistent). We can remove any of A, B, or $A \wedge B \to C$, but which one should we choose?

The key question in belief revision, then, is how to define the revision and contraction functions so that they choose the appropriate formulas to give up. Alchourrón et al. [1985] considered this question in some detail, defining a list of postulates that the new set of formulae (e.g., Δ^*) should follow. Among other things, these postulates stipulate what a rational procedure for determining the new set should do, including ensuring that the new data is in the revised set.

This rational procedure then defines a partial order over the formulas in Δ, called the *epistemic entrenchment*, which defines the order in which we should abandon the original formulas. For example, we might say that B was learned before A, so remove it first (on the grounds that we ought to keep the most recently learned formulas as these are more likely to be true).

We return to the concepts in this section when we discuss the problem of revising requirements, in Chapter 8.

2.3 Knowledge Level Models

The idea of knowledge level modeling captures, in general, the distinction between a specification of behaviour and the implementation of that behaviour. This is by now a fairly well-understood idea: object-oriented programming concepts such as abstraction and information-hiding are directly related. One early example of this is the classic paper by Parnas [1978] which introduces interfaces to disintermediate between a general solution and specific implementations thereof.

In knowledge representation, Levesque [1981] gave an early use of the idea of specifying operations on the knowledge base abstractly, using a language *KL* to describe queries on the knowledge base, in terms of the functions the knowledge base could perform, as distinct from the symbol-level implementation of those functions. Knowledge-level modeling was popularized by Newell [1982], who made it clear that knowledge "is to be characterized functionally, in terms of what it does, not structurally in terms of physical objects with particular properties and relations". The advantage of this is the same as for abstract data types (ADTs) in software engineering: the ability to associate different implementations with a single functional definition; and the ability to hide from the user of that data type the details of how it works.

Levesque [1984] gives a canonical example of a Stack ADT. Stacks define functions pop() and push() to remove and add elements from the stack, respectively; the specifics of how to return an element (e.g., using a circular array) are left to the implementation. In this thesis, a knowledge-level specification provides **a)** a list of *operations* with their syntactic signatures, and **b)** an implementation-independent *specification* of the effect of each operation. These are then completed by an implementation.

In requirements engineering, the knowledge-level account has been used in the description of the Telos language [Mylopoulos et al., 1990]. Telos defined a knowledge representation language designed to "support software engineers in the development of information systems throughout the software lifecycle." The language specified a functional interface using a formal notation and a means of drawing inferences from a knowledge base. To add information, the language defined operations TELL, UNTELL and RETELL, which allow one to add, remove and update the information in the knowledge base, respectively (RETELL is the same as UNTELL and then TELL). To extract information we can use RETRIEVE, which limits the complexity of the inferences the knowledge base supports, or ASK, which is a general-purpose query. This separation between adding and querying the knowledge base, and defining languages for each operation, is one to which we shall return in Chapter 6.

Chapter 3

Related Work

In this section I review work which is related to the rest of my thesis. I begin by describing the historical context behind requirements evolution (§3.1). Industrial tools generally do not manage changing or evolving requirements well, but §3.2 describes what industry considers state of the art. After looking at implementation level approaches to changing requirements (§3.3), §3.4 surveys academic empirical studies of evolving or changing requirements. The selection criteria is that the study must be based on artifacts used in practice. This chapter concludes with §3.5, a survey of current state of the art in research.

3.1 Historical Developments

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 portion tries to understand how changes in requirements impact software maintenance.

The study of software evolution begins with work by 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 Lehman and Fernández-Ramil [2006]. These papers characterized the nature of software evolution as an inevitable part of software development. This inevitability implied that programs must continually be maintained to minimize distance from the constantly occurring changes in the operating domain. One law states that software quality will decline unless regular maintenance activity happens, and another implies that as a consequence of these changes a system will increase in complexity with time. While their work largely focused on implementation artifacts, it clearly identifies requirements as driving the corrective action necessary to reconcile actual with anticipated behavior: “Computing requirements may be redefined to serve new uses [Belady and Lehman, 1976, p. 239].”

Early on in the history of software development it became clear that building software systems was nothing at all like engineering physical structures. 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 [Basili and Perricone, 1984]). Swanson [1976] 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 identifies “changes in data and processing environments” as a major cause of *adaptive* maintenance activity. I examine these changes in more detail later on. 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, 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. Following from this was the slightly more flexible software process standard IEEE/ISO–12207, and IEEE–830, perhaps the most well-known standard for software requirements to date. But David Parnas’s paper on the “Star Wars” missile defence scheme [Parnas, 1985] illustrated the problems with this approach, many of which come down to an inability to anticipate all 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 [Parnas, 1985, p. 1329].” This demonstrated the massive impact unanticipated change could have on software systems, a central concern of this thesis. Indeed, the US military no longer insists that software be developed according to any standard in particular [McDonald, 2010, p. 42].

In response to the problems with the waterfall approach, iterative models, such as Boehm [1988]’s ‘spiral’ model of development called for iterations over system design, such 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., Ambler [2006]). 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 [Brooks, 1975].” 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 Cockburn [2008].

Requirements Evolution — This thesis focuses on that part of software evolution that is caused by changing requirements or assumptions (i.e., the components of the requirements problem which are in G or D). 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 et al. [2001, p. 17] concludes that evolution only occurs when maintenance impacts business rules or changes properties visible to the customer.

Harker et al. [1993] extended Swanson’s work to focus on change drivers with respect to system requirements (summarized in Table 3.1), because “changing requirements, rather than stable ones, are the norm in systems development [Harker et al., 1993, p. 266].” He characterized changes according to what their origins were. 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 3.1: Types of requirements change [Harker et al., 1993]

	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 question whether there is there such a thing as an enduring requirement, as Harker et al. define. A useful analogy can be derived from Stuart Brand’s book on architectural change in buildings [Brand, 1995]. 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 in our 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 crack the security model. So I posit that the enduring/changing distinction originates in the abstractness of the requirement, rather than any particular characteristic. In my thesis, I have eschewed discussions about change-proneness in favour of a general framework for any changing requirement.

This thesis looks, for the most part, at what Harker calls 'consequential' requirements, because my focus is on systems that exist, and what happens when those requirements change. The trouble with Harker's taxonomy is that it does not allow for a requirement to occupy multiple categories. For example, designers often deal with requirements which are discovered after the system is implemented, but which are not caused by that system. Instead, it reflects a property of the environment – the requirement addresses something mutable.

Harker's taxonomy was expanded by the EVE project [Lam and Loomes, 1998]. They emphasized that requirements evolution is inevitable and must be managed by paying attention to four 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, e.g., [Basili and Perricone, 1984]. Somerville and Sawyer's requirements textbook [Sommerville and Sawyer, 1997] 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 [Bubenko, 1980] 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 [Bubenko, 1980, p.401]." Little guidance was given on how to do this, however. In a similar vein, RML [Greenspan et al., 1982], ERAE [Dubois et al., 1986] and Telos [Mylopoulos et al., 1990] gave validity intervals for model instances using temporal logic. 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. By contrast, my thesis proposes using these models for a prolonged period of time over several system iterations.

Research has also looked at 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 in Easterbrook and Nuseibeh [1995]. The inconsistency was managed using abduction in Menzies et al. [1999], which I extend in Chapter 8. Multiple-valued logics, which my colleagues and I have considered, were used to combine inconsistent viewpoints in Chechik and Easterbrook [2001]. Inconsistency in evolving requirements can also occur as they are elaborated. Zowghi and Offen [1997] applied a default logic reasoning approach to maintaining consistent sets of requirements during revision and expansion; I shall return to this idea in Chapter 8. Similarly, Ghose [2000] used logic to manage evolving requirements, defining a revision choice function to determine which requirements to maintain in the new version.

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 Antón [1996]); in this thesis I focus on models which have already been elaborated, and now change, rather than the process of requirements elicitation at a point in time.

3.2 Evolution and Industrial Tools

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 my 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 (for good or for ill). Furthermore, these tools are not the whole story, as many industry consultants (e.g., Wiegers [2003]; Leffingwell and Widrig [2003]) 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.

Standards and industry – IEEE Software Engineering Standards Committee [1998], which describes a standard for "Software Requirements Specification" (SRS) is the culmination of this 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 deals with evolution, which it recommends managing change 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.

Requirements management tools – Commercial tools do a poor job supporting change. IBM DOORS[1] 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[2] is agile, with strong support for simulation and prototyping. Workbenching requirements scenarios is important in Blueprint, something I address in coming chapters. 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.

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[3], 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 [Anderson, 2010] 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.

Particularly for smaller organizations, requirements are not treated at a high-level, often existing as an Excel spreadsheet or maintained implicitly [Aranda et al., 2007]. Furthermore, the transition to agile software development has made one of its chief priorities the reduction of unnecessary documentation

[1]http://www-01.ibm.com/software/awdtools/doors/
[2]http://www.blueprintsys.com/products/
[3]http://www.atlassian.com/software/jira/

("working software over comprehensive documentation"[4]). 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. This is the approach that we take in this thesis – that a tool which can preserve the overall requirements model throughout the lifecycle is necessary. That is not to say it such an approach could not be integrated into a tool like IBM DOORS.

3.3 Implementation-level Approaches

Work on unanticipated software evolution has been approached from the implementation level. The focus is on replacing one version of a component with the new version (often called *dynamic object evolution*). To use the example from Kniesel [1999], when the Euro was introduced, no banking software anticipated such a development, and therefore merely 'adapting' software behaviour was not possible. In addition, most banking software is not amenable to extended downtime. Research has therefore examined approaches which could replace 'running' code (e.g., replace bytecode in a virtual machine). Dyer and Rajan [2010], for example, use dynamic aspect-oriented programming techniques to insert new code into a running implementation.

It also seems likely that similar approaches would be feasible in service oriented computing. The wrapper approach (also known as the Adapter pattern) has been applied in e.g., Kaminski et al. [2006]. The intermediate representation adapts the higher-level operation into one which supports the current version of the functionality (e.g., Euro or Deutschmark currency). OSGI, for one, has demonstrated a Java-based technology for run-time bundle switching, based on simple specifications of capabilities [OSGi Alliance, 2009].

While useful, these approaches do not provide a way to address system goals in the form of requirements. I feel these techniques are complementary to the work in this thesis, but I do not examine in great detail the important transition from requirements to implementation. The assumption is that having been told what implementation tasks are required, the designers can then proceed to subsequent software engineering phases, such as architecture and design.

[4]http://agilemanifesto.org/

3.4 Studies of Evolving Requirements in Industrial Settings

This section motivates the importance of the Requirements Evolution Problem (REP) by showing other academic studies addressing the issue. In particular, here I focus 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. Alspaugh et al. [1992] is a retrospective of the project, including needed 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 greatly detailed, and some invariants are assumed (which I would term domain assumptions, such as "weapon delivery cannot be accurate if location data is not accurate").

Chung et al. [1996] 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.

Antón and Potts [2001] 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 effort of providing a new service to customers. This survey was end-user oriented; by contrast, I approach the problem of feature evolution from the internal perspective: how the external forcings (e.g., customer demand) force changes to requirements.

Anderson and Felici [2000, 2001], conducted a series of case studies of changing requirements. Their experiences informed 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.

As I do in Chapter 4 , Tun et al. [2008] uses 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. Chapter 4 shows that extracting *quality* requirements, which are domain-independent, was in fact possible.

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? Wnuk et al. [2009] looked at this issue in an industrial case study, as did Xue et al. [2010].

Herrmann et al. [2009] used an industrial case study to identify the need for "delta requirements", requirements which must be added subsequent to software delivery. This approach to specifying a delta from the existing requirements is mirrored in my work. We both highlight the importance of incremental refinements of requirements; my approach is more theoretical and focuses on automating the discovery of new deltas. Herrmann et al. took an existing methodology and extended it to incorporate modeling techniques for delta requirements.

My final example of industrial case studies comes from Navarro et al. [2010], which looked at the issue of changing requirements in the highly formal requirements environment of spacecraft design. There, the issue is to minimize 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.

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. In the remaining chapters, I will propose my framework for addressing these concerns by minimizing the number of changes to be made as the requirements change.

3.5 Current Approaches

Let us examine some of the current research initiatives focusing on the problem of changing requirements.

Anticipated changes — Spurred, perhaps, by the seminal paper of Kephart and Chess [2003], the majority of research has focused on managing anticipated changes. Kephart and Chess [2003]'s idea was to force the software to take on more of the reconfiguration tasks. A simple reconfiguration task might be to allocate more RAM when a server reaches a certain load. The reasoning behind this initiative is that in most cases, reconfigurations were simple and tractable to specify. Accordingly, research has looked at ways in which this reconfiguration can be monitored, analyzed, planned, and executed. Most of these changes are assumed to occur in the environment. This might involve a change in location [Ali et al., 2008; Dalpiaz et al., 2009] or load factors for databases [Duez and Vicente, 2005]. Similarly, work on monitoring requirements has focused on anticipating possible re-configurations: Wang and Mylopoulos [2009], for example, looked at reconfiguration for monitored goal models: if one task fails then the system searches for a viable alternative configuration. With the RELAX framework [Whittle et al., 2009], a language is specified which can adapt flexibly to run-time changes in requirements. In a similar vein, fuzzy goals use fuzzy probabilistic logic to guide system responses [Baresi et al., 2010]. Research has also looked at using control variables in conjunction with awareness requirements to dynamically adjust to new contexts [Silva Souza et al., 2011].

Unanticipated changes — Changes are not always anticipatable, however, and often these changes are the most important. For example, business is at the mercy of legislative whims of governments. A new piece of privacy legislation may require drastic changes in software systems, e.g., to remove personal information from databases. Thus, assuming in your framework that you can build in all possible reconfigurations is naive. Nanda and Madhavji [2002] point out the importance of environmental evolution on the requirements of an existing system. One approach to this is to re-use requirements, as I discuss in Chapter 7 , which was the focus of Herrmann et al. [2009]. Another is to allow for impact analysis which can pinpoint the result of making a change. One approach, which does not allow for model changes, is used in the AGORA tool [Tanabe et al., 2008].
Inverardi and Mori [2010] describe a feature engineering approach to anticipated change which mirrors my work (in Chapter 6) on maintaining a consistent set of features, which, in accordance with this thesis, are 3-tuples of $\langle$ Domain assumptions, Requirements, Specification $\rangle$, as defined in Classen et al. [2008]. However, the operations are not described in much detail, and no analysis of complexity is performed. Still, this paper is important in identifying the distinction between (as they term it) 'foreseen and unforeseen' evolution (which corresponds to *anticipated* and *unanticipated* in my terminology). The way of doing this in their work is to co-evolve requirements and tests to preserve requirements relevance.

Traceability — Traceability refers to the ability to describe and follow the life of a requirement [Gotel and Finkelstein, 1994]. Research in requirements traceability is also concerned with changing requirements, although the focus is on maintaining traceability between requirements and specification, rather than the nature of the change itself. Accurate traceability between requirements and implementation remains fairly elusive. Rather than focusing on traceability directly, I assume the existence of such links, and concentrate on the inter-relationship of requirements, specification, and domain assumptions—the requirements problem.

Chapter 4

Empirical Studies of Changing Requirements

This chapter conducts empirical studies of requirements in industrial software projects in order to motivate the study of a Requirements Engineering perspective on software evolution. There have been objections to making requirements more prominent in the study of software evolution. This chapter will show that these concerns are misplaced. I do so using some empirical results that demonstrate that the Requirements Engineering perspective is useful. In many ways it is primary, particularly when one deals with higher-level software artifacts such as design documents or standards.

The primary objections are the following:

1. For many people, source code is the only accurate artifact [Harman, 2010]. Requirements are either not used explicitly, or are stale the moment they are 'completed'.
2. In terms of quantity, most change tasks involve low-level corrective maintenance tasks, rather than more complex adaptive maintenance. Therefore, the objection goes, efforts are better focused there.

One can marshal the following rebuttals:

1. The problem of requirements relevance is a well-documented concern. Research has shown that traceability from requirements to code is possible, as is requirements recovery, but both are still too difficult in practice [Ben Charrada and Glinz, 2010]. In many cases, such as larger-scale projects or projects tackling 'wicked' problems, useful and relevant requirements documents *do*

exist. Nonetheless, this is an important area for more research, of concern to the wider field of model-driven engineering as well.

2. As for the second objection, it may seem as though the numbers of corrective change tasks are more numerous. For example, if you examine widely-used task management repositories (such as Mozilla's BugZilla), by far the most common task is a bug report: something crashed my browser, and I would like you to fix it. However, the literature, including [Swanson, 1976; Hindle et al., 2008], does not support this contention. My position is that the adaptive or perfective changes implied by requirements evolution are more complex and more costly, and therefore more important, than focusing on corrective bug fixes.

This chapter will address these points in two ways. First, I present work on empirical studies of quality requirements (§4.1). This will help to establish that the phenomenon of changing requirements exists; that requirements, in this case quality requirements such as usability and maintainability, react to external drivers, and in turn drive activity in the source code. Furthermore, this study shows that a one-size-fits-all approach is unlikely to be successful. Projects in the study focused on different qualities in their development activities.

Secondly, I present the results of a case study of requirements change in distributed computing specifications (§4.2). In this study, I examined a longer-term change in the domain assumptions underlying the introduction of distributed computing technologies (standards and specifications such as CORBA, DCOM and REST). This study highlighted how previously unanticipated requirements and domain assumptions (such as bandwidth limits) affected later technologies.

I conclude by analyzing the lessons these empirical studies present for understanding the requirements evolution problem.

4.1 Mining Quality Requirements From Artifacts

A central conjecture in this thesis is that requirements changes drive software changes. The role of this section is to support this conjecture by showing that this causal relationship does exist. I do this with two studies looking at software development artifacts including source code commit comments (that is, the comments made when adding changed source code to a repository), mailing list messages, and bug reports. The open-source software (OSS) community has made examining these artifacts much simpler in recent years, because there are no access restrictions on the data, unlike with most for-profit companies.

The key challenge in these studies was to extract the implicit requirements, since there is no traceability data from original requirements documents, such as they were. If the requirements problem can be defined in Techne as $D, S \vdash G$, then requirements extraction is discovering the domain assumptions and specification and reverse-engineering the goals from that. In this chapter, I focus on finding information about the domain assumptions as described in project artifacts, and use this to understand, statistically, the possible goals. This requires error analysis of the extraction mechanisms.

The central insight behind this effort was that for a certain class of requirements it was possible to make cross-project comparisons. This class were the non-functional requirements, also known as software qualities. I leveraged the ISO9126 standard [Coallier, 2001] for describing software quality. I reasoned that this taxonomy applies to all software projects (e.g., presumably, to a greater or lesser degree, every project is concerned with *usability*), and I could therefore use it to compare projects.

There are two studies that I conducted: one, published in [Ernst and Mylopoulos, 2010], discussed in §4.1.1 looked at eight projects and attempted to identify a correlation between requirements and project lifespan. The motivation for this was the notion that software projects must constantly and increasingly focus on maintenance as the project ages. This is known as Lehman's Seventh Law [Lehman et al., 1997]. More importantly, I was able to associate changes in the focus on a particular quality with changes that occurred in the external environment.

The second study (§4.1.2), jointly conducted with Abram Hindle and Mike Godfrey and published in [Hindle et al., 2011], focused on statistical techniques to mine quality requirements from source code commits. This work is more technical, and concerned with showing that labelling topics with quality requirements is possible.

4.1.1 Controlled Vocabulary Mining of GNOME Projects

Quality requirements are usually defined in terms of global properties for a software system, such as "reliability", "usability" and "maintainability"; we think of them as describing the 'how', rather than the 'what'. In this sense "functionality" can also be considered a quality, insofar as it describes how well a given artifact implements a particular function (such as security). The importance of quality requirements lies in their inter-system comparability. One approach to these requirements is to treat them as non-functional requirements (NFRs) [Chung et al., 1992].

If requirements are important throughout the lifecycle, a better understanding of requirements after the initial release is important. Are requirements discussed post-release? One way of answering this question is to examine current practices using a standardized requirements taxonomy. In particular, we are interested in finding out whether there is any noticeable pattern in how software project participants conceive of quality requirements. The study is conducted from the perspective of project participants (e.g., developers, bug reporters, users).

I used a set of eight open-source software (OSS) products to test two specific hypotheses about software quality requirements. The first (null) hypothesis is that software quality requirements are of more interest as a project ages, as predicted in Lehman's 'Seventh Law' – that "the quality of systems will appear to be declining unless they are rigorously maintained and adapted to environmental changes [Lehman et al., 1997, p. 21]." The second (null) hypothesis is that quality is of similar concern among different projects. That is, is a quality such as *Usability* as important to one project's participants as it is to another. Testing this hypothesis will give some idea of whether requirements are driving changes in software development. If there is no difference, if the null hypothesis is correct, we might conclude that quality requirements, at least, are irrelevant. On the other hand, a difference suggests that these concepts, even if latent, are a concern to the project.

To assess these questions, I need to define what I mean by software quality requirements. My position is that requirements for software quality can be conceived as a set of labels assigned to the conversations of project participants. These conversations take the form of mailing list discussions, bug reports, and commit logs. Consider two developers in an OSS project who are concerned about the software's performance. To capture this quality requirement, I look for indicators, which I call **signifiers**, which manifest the concern. I then label the conversations with the appropriate software quality, using text analysis. These qualities are derived from a standard taxonomy – the ISO 9126–1 software quality model [Coallier, 2001]. The signifiers are keywords that are associated with a particular quality. For example, I label a bug report mentioning the **slow** response time of a media player with the *Efficiency* quality.

The subsection on methodology, following, describes how I derive these signifiers and how I built the corpora and toolset for extracting the signifiers. I then present the observations and a discussion about significance in the subsection on observations.

Methodology

Overview: I first construct a set of signifiers, which produces a word list to extensionally define the software quality of interest, e.g., *Efficiency.* I then query corpora from each project with these lists to identify events. Events are timestamped occurrences of signifiers in the corpora. Figure 4.1 gives a visual overview.

Step 1 - Establishing the Corpora The corpora are from a selection of eight Gnome projects, listed in Table 4.1. Gnome is an OSS project that provides a unified desktop environment for Linux and its

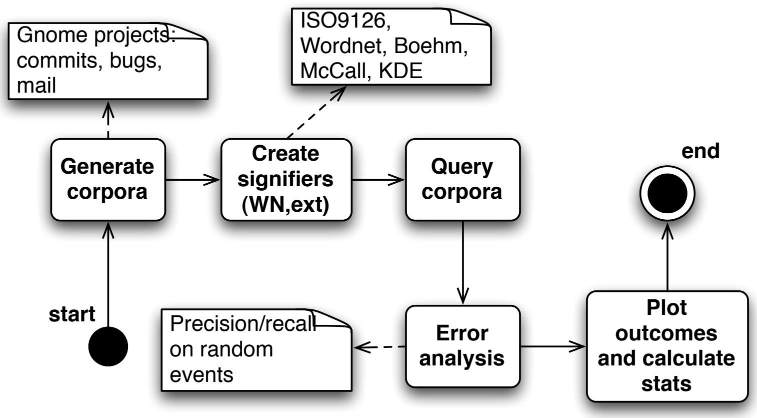

Gnome projects: commits, bugs, mail
ISO9126, Wordnet, Boehm, McCall, KDE
end
Generate corpora
Create signifiers (WN,ext)
Query corpora
start
Precision/recall on random events
Error analysis
Plot outcomes and calculate stats

Figure 4.1: Overview of methodology.

Table 4.1: Selected Gnome ecosystem products (*ksloc* = thousand source lines of code)

Product	**Language**	**Size** *(ksloc)*	**Age** *(years)*	*Type*
Evolution	`C`	313	10.75	Mail
Nautilus	`C`	108	10.75	File mgr
Metacity	`C`	66	7.5	Window mgr
Ekiga	`C++`	54	7	VOIP
Totem	`C`	49	6.33	Media
Deskbar	`Python`	21	3.2	Widgets/UI
Evince	`C`	66	9.75	Doc viewer
Empathy	`C`	55	1.5	IM

cousins. Gnome is both a project and an ecosystem: while there are co-ordinated releases, each project operates somewhat independently. In 2002, Koch and Schneider [2002] listed 52 developers as being responsible for 80% of the Gnome source code. In this study, the number of contributors is likely higher, since it is easier to participate via email (e.g., feature requests) or bug reports. For example, in Nautilus, there were approximately 2,000 people active on the mailing list, whereas there were 312 committers to the source repository. [1]

The projects used were selected to represent a variety of lifespans and codebase sizes (generated with [Wheeler, 2009]). All projects were written in C/C++, save for one in Python (Deskbar). For each project I created a corpus from that project's mailing list, subversion logs and the bug comments, as of

[1]Generated using Libresoft, tools.libresoft.es

November 2008. From the corpus, I extracted 'messages', that is, the origin, date, and text (e.g, the content of the bug comment), and placed this information into a MySQL database. A message consists of a single bug report, a single email message, or a single commit. If a discussion takes place via email, each individual message about that subject is treated separately. The dataset consists of over nine hundred thousand such messages, across all eight projects.

I cleaned the data of auto-generated emails or messages (e.g., "This bug is a duplicate of bug #3343") and auto-generated stack traces (such as those produced by BugBuddy on Linux or crash reporters in Firefox). Finally, I eliminated those events which were due to administrative tasks (e.g., moving bug repositories results in a large number of status changes unrelated to any particular requirements).

Note, I do not extract information on the mood of a message: i.e., there is no way to tell whether this message expressed a positive attitude towards the requirement in question (e.g., "This menu is unusable"). Furthermore, I am not linking these messages to the implementation in code; there is no way of telling to what extent the code met the requirement beyond participant comments.

Table 4.2: Qualities and quality signifiers – Wordnet version (WN). Bold text indicates the word appears in ISO/IEC 9126.

Quality	**Signifiers**
Maintainability	**testability changeability analyzability stability maintainability** maintain maintainable modularity modifiability understandability
Functionality	**security compliance accuracy interoperability suitability** functional practicality **functionality**
Portability	**conformance adaptability replaceability installability** portable movableness movability **portability**
Efficiency	**"resource behaviour" "time behaviour"** efficient **efficiency**
Usability	**operability understandability learnability** useable usable serviceable usefulness utility useableness usableness serviceableness serviceability **usability**
Reliability	**"fault tolerance" recoverability maturity** reliable dependable responsibleness responsibility reliableness **reliability** dependableness dependability

Step 2 - Defining Qualities with Signifiers In semiotics, C. S. Peirce drew a distinction between signifier, signified, and sign [Atkin, 2006]. In this work, I make use of signifiers – words like 'usability' and 'usable' – to capture the occurrence in the corpora of the signified – in this example, the concept *Usability.* I extract the signified, concept words from the ISO 9126 quality model [Coallier, 2001], which describes six high-level quality requirements (listed in Table 4.2). There is some debate about the significance and importance of the terms in this model. However, it is "an international standard and thus provides an internationally accepted terminology for software quality" Boegh [2008, p.58], which is sufficient for the purposes of this research.

The model is used to refer to both external and internal views of quality (for example, bugs filed by users would be external qualities, whereas an email discussion of features to come would be internal).

I want to preserve domain-independence, in order to use the same set of signifiers on different projects. This is why I eschew more conventional text-mining techniques that generate keyword vectors from a training set.

I generate the initial signifiers from Wordnet [Fellbaum, 1998], an English-language 'lexical database' that contains semantic relations between words, including meronymy and synonymy. I extract signifiers S for T, the set of top-level terms in ISO9126–1 as follows:

$S = \bigcup_{t \in T}$ spellings(syns$(t) \cup$ hyper$(t) \cup$ meron$(t) \cup$related(t))

Wordnet's synsets *syn*, hypernyms (*hyper*), and related forms (stemming, *meron*), and related components *rel* is done using the two-level hierarchy in ISO9126. After accounting for spelling variations *spell*, I then associate this wordlist with a top-level quality, and use that to find unique events. This gives us a repeatable procedure for each signified quality. I call this initial set of signifiers `WN`.

Step 3 - Querying the Corpora Once I constructed the sets of signifiers, I applied them to the message corpora (the mailing lists, bug trackers, and repositories) to create a table of events. An *event* is any message (row) in the corpus table which contains at least one term in the signifier set. A message can contain signifiers for different qualities, and can this generate as many as six events (e.g., a message about maintainability and reliability). However, multiple signifiers for the *same* quality only generate a single event for that quality. I produced a set of events (e.g., a subversion commit message), along with the associated time and project. Events are grouped by week for scalability reasons. Note that each email message in a thread constitutes a single event. This means that it is possible that a single mention of a signifier in the original message might be replied to multiple times. I assume these replies are 'on-topic' and related to the original concern.

I normalize the extracted event counts to remove the effect of changes in mailing list volume or commit log activity (some projects are much more active). The calculation takes each signifier's event count for that period, and divides by the overall number of messages in the same period. I also remove low-volume periods from consideration. This is because a week in which only one message appeared, that contained a signifier, will present as a 100% match. From this dataset I conducted the observations and statistical tests. Table 4.3 illustrates some of the sample events dealt with, and the subsequent mapping to software quality requirements.

Table 4.3: Classification examples. Signifiers causing a match are highlighted.

Event	**Quality**
...By upgrading to a newer version of GNOME you could receive **bug** fixes and new functionality.	*None*
There should be a feature added that allows you to keep the current **functionality** for those on workstations (automatic hot-sync) and then another option that allows you to manually initiate .	*Functionality*
Steps to reproduce the **crash**: 1. Can't reproduce with **accuracy**. Seemingly random.	*Reliability, Functionality*
How do we go disabling ekiga's **dependency** on these functions, so that people who arn't using linux can build the program without having to resort to open heart surgery on the code?	*Maintainability*
U_() is equivalent of _() but returns **Unicode** (UTF-8) string. Update your xml-**i18n**-tools from CVS (recent version understands U_), update Swedish **translation** and close the bug back.	*Portability*
On some thought, centering **dialogs** on the panel seems like it's probably right, assuming we keep the **dialog** on the screen, which should happen with latest metacity.	*Usability*
These calls are just a waste of time for client and server, and the Nautilus online storage view is **slowed** down by this wastefulness.	*Efficiency*

Step 4 - Expanding the Signifiers The members of the set of signifiers will have a big effect on the number of events returned. For example, the term 'user friendly' is one most would agree is relevant to discussion of usability. However, this term does not appear in Wordnet. To see what effect an expanded

Table 4.4: Qualities and quality signifiers – extended version (‘ext‘). Each quality consists of terms in (a) in addition to the ones listed.

Quality	**Signifiers**
Maintainability	WN + interdependent dependency encapsulation decentralized modular
Functionality	WN + compliant exploit certificate secured ”buffer overflow” policy malicious trustworthy vulnerable vulnerability accurate secure vulnerability correctness accuracy
Portability	WN + specification migration standardized l10n localization i18n internationalization documentation interoperability transferability
Efficiency	WN + performance profiled optimize sluggish factor penalty slower faster slow fast optimization
Usability	WN + gui accessibility menu configure convention standard feature focus ui mouse icons ugly dialog guidelines click default human convention friendly user screen interface flexibility
Reliability	WN + resilience integrity stability stable crash bug fails redundancy error failure

list of signifiers would have, I generated a second set (henceforth `ext`), by expanding `WN` with more software-specific signifiers. The `ext` signifier sets are shown in Table 4.4.

To construct the expanded sets, I first used Boehm’s 1976 software quality model [Boehm et al., 1976], and classified his eleven ‘ilities’ into their respective ISO9126 qualities. I did the same for the quality model produced by McCall [1977]. Finally, I analyzed two mailing lists from the KDE project to enhance the specificity of the sets. Like Gnome, KDE is an open-source desktop suite for Linux, and likely uses comparable terminology. I selected KDE-Usability, which focuses on usability discussions for KDE as a whole; and KDE-Konqueror, a list about a long-lived web browser project. For each high-level quality in ISO9126, I first searched for the existing (WN) signifiers; I then randomly sampled twenty-five mail messages that were relevant to that quality, and selected co-occurring terms relevant to that quality. For example, I add the term ‘performance’ to the synonyms for *Efficiency*, since this term occurs in most mail messages that discuss efficiency.

There are other possible sources for quality signifiers. I discuss the differences the two sets create in the next section.

Step 5 - Precision and Recall The use of indicator words is shallow, and so easy to confuse. Consider the text “slow acceptance by users”. In this case, most probably the message is about *Usability* rather than the *Efficiency* signifier that I listed originally. I verified the percentage of terms retrieved that were unrelated to a signified software quality to understand the precision of the approach. For example, I encountered some mail messages from individuals whose email signature included the words "Usability Engineer". If the body of the message wasn't obviously about usability, I coded this as a false-positive. The error test was to randomly select messages from the corpora and code them as relevant or irrelevant. I assessed 100 events per quality, for each set of signifiers (`ext` and `WN`). For the *Nautilus* project, this random sample represents 3.5% of the total retrieved matching events for "Usability". A more stratified approach might sample based on project participant or time. Table 4.5 presents the results of this test. False-positives averaged 21% and 20% of events, for `ext` and `WN` respectively (i.e., precision was 79% and 80%).

Recall, or completeness, is defined as the number of relevant events retrieved divided by the total number of relevant events. Superficially I could describe recall as 100%, since the query engine returns all matches it was asked for, but true recall should be calculated using all events that had that quality as a topic. To assess this, I randomly sampled the corpora and classified each event into either a signifier (*Usability*, *Reliability*, etc.) or *None*. The extended signifier lists had an overall recall of 51%, and a poor 6% recall for the Wordnet signifiers. I therefore dispensed with the Wordnet signifier set. This is a very subjective process. For example, I classified a third of the events as *None*; however, arguably any discussion of software could be related, albeit tangentially, to an ISO9126 quality. A better understanding of this issue is more properly suited to a qualitative study, in which project-specific quality models can be best established.

Table 4.5: False positive rates for the two signifier sets

Signified quality	F.P. Rate ext	F.P. Rate WN
Usability	0.47	0.22
Portability	0.11	0.20
Maintainability	0.22	0.31
Reliability	0.15	0.19
Functionality	0.14	0.18
Efficiency	0.16	0.07
Mean	**0.21**	**0.20**

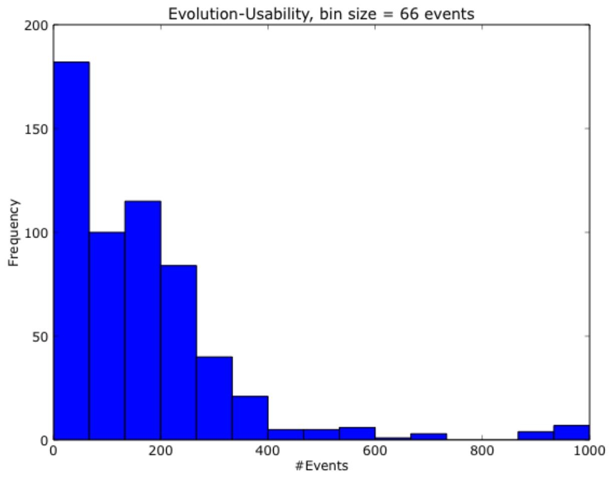


Figure 4.2: Frequency distribution for Evolution-Usability. *x-axis* represents number of events (66 events wide), *y-axis* the number of weeks in that bin.

Observations

This section first explains the frequency distributions of the data I collected. That data is then used to answer the two hypotheses raised in the §4.1.1:

1. There a correlation between discussion of quality requirements and project age.

2. Quality requirements of similar importance relative to each project.

Data distribution Fig. 4.2 shows an example frequency distribution for the quality requirement *Usability*, product *Evolution*, with non-normalized data. The distributions follow a power-law distribution, that is, a majority of weeks have few events, with the 'long tail' consisting of those weeks with many events. For example, the left-most bin contains over 175 weeks with between 0 and 66 events. I verified that this power-law pattern also existed for the remaining qualities and project combinations, but do not display them for brevity.

Examining quality discussions over time The null hypothesis was that, as predicted in the literature, there should be a correlation between the importance of software quality requirement and the age of a project. I examined this in three ways. First, I looked at the overall trends for a project. Secondly, I used release windows, the time between the release of one version, and the release of the next (major) version. Given the negative results, I explored qualitative explanations for patterns in the data.

Using project lifespan – I examined whether, over a project's complete lifespan, there was a correlation with quality event occurrences. Recall that I define quality events as occurrences of a quality

Table 4.6: Selected summary statistics, normalized. Examples from Nautilus and Evolution for all qualities using extended signifiers.

Project	Quality	r^2	slope	N (weeks)
Evolution	Efficiency	0.06	-0.02	439
	Portability	0.08	-0.05	448
	Maintainability	0.04	-0.02	320
	Reliability	0.20	0.25	492
	Functionality	0.03	-0.02	439
	Usability	0.14	0.27	515
Nautilus	Efficiency	0.16	-0.10	420
	Portability	0.16	-0.07	331
	Maintainability	0.27	-0.09	216
	Reliability	0.19	0.26	454
	Functionality	0.12	-0.05	390
	Usability	0.08	0.29	459

signifier in a message in the corpora. I performed a linear regression analysis and generated correlation coefficients for all eight projects and six qualities. Figure 4.3 is an example of the analysis. It is a scatterplot of quality events vs. time for the *Usability* quality in Evolution. For example, in 2000/2001, there is a cluster around the 300 mark, using the extended (`ext`) set of signifiers. Note that the y-axis is in units of (events/volume * 1000) for readability reasons.

The straight line is a linear regression. The dashed vertical lines represent Gnome project milestones, with which the release dates of the projects under study are synchronized. Release numbers are listed next to the dashed lines. Table 4.6 lists only Nautilus and Evolution as products, and r^2 — squared correlation value, or coefficient of determination – and slope (trend) values for each quality within that project. r^2 varies between 0 and 1, with a value of 1 indicating perfect correlation. The sign of the slope value indicates direction of the trend. A negative slope would imply a decreasing number of occurrences as the project ages. Table 4.7 does a similar analysis for all products and the *Usability* and *Efficiency* (performance) qualities.

The results show no correlation, and thus support rejecting the null hypothesis. In all cases the correlation coefficient – indicating the explanatory power of the linear regression model – is quite low,

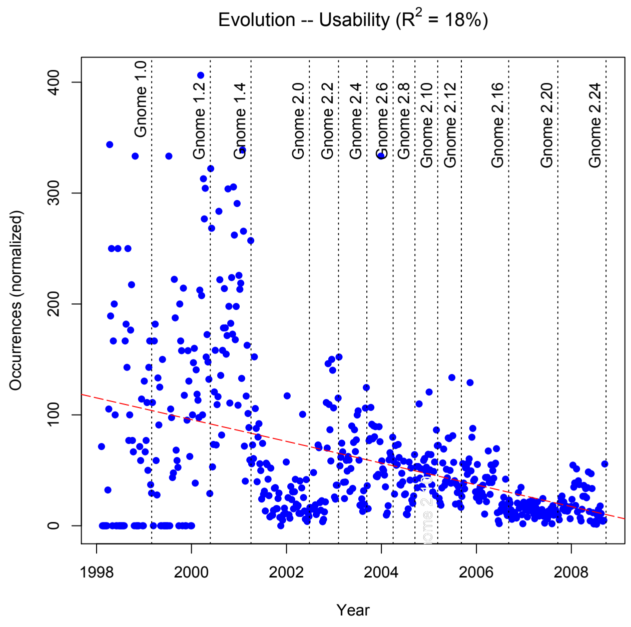


Figure 4.3: Signifier occurrences per week, Evolution – *Usability*

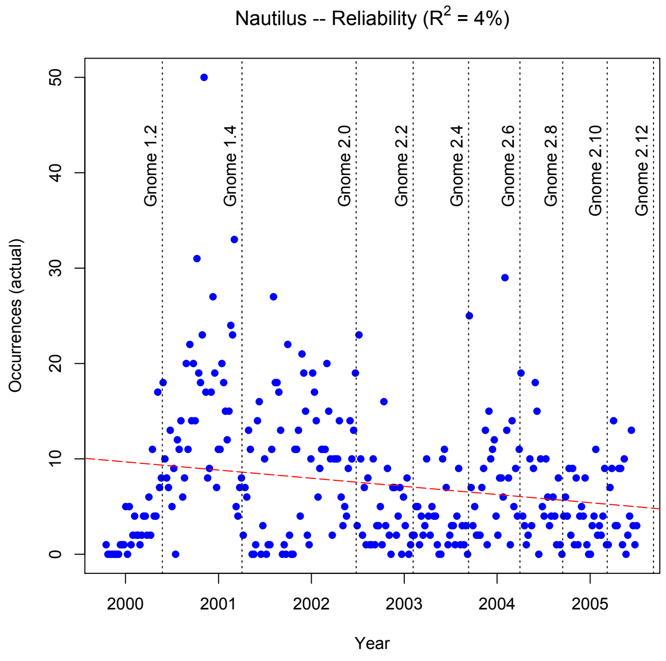


Figure 4.4: Signifier occurrences per week, Nautilus – *Reliability*

well below the 0.9 threshold used in, for example, [Mens et al., 2008]. There does not seem to be any reason to move to non-linear regression models based on the data analysis I performed. I conclude that the extended list of signifiers does not provide any evidence of a relationship between discussions of software quality requirements and time. In other words, either the occurrences of the signifiers are random, or there is a pattern, and the signifier lists are not adequately capturing it. The former conclusion seems more likely based on inspection of the data.

Using release windows – It is possible that the event occurrences are more strongly correlated with time periods prior to a major release, that is, that there is some cyclical or autocorrelated pattern in the data. I defined a release window as the period from immediately after a release to just before the next release. I then investigated whether there was a higher degree of correlation between the number of quality events and release age, for selected projects and keywords. This release window correlation was no better than the one found for project lifespan as a whole. In other words, there is no relationship between an approaching release date and an increasing interest in software quality requirements.

Table 4.8 shows the results for selected products and releases (starting with release 2.8, in which Evolution was synchronized with Gnome). I used the data from the extended signifier list (`ext`). For release 2.8, there were 27 weeks between releases, and for release 2.22, 28 weeks. For most release dates, there is no improvement in the explanatory power of the model.

Analysis of key peaks in selected graphs – To explain why the null hypothesis, that there ought to be correlation, I theorized that the data are unrelated to software age or release cycle, and are instead responding to external events, such as a usability audit. I chose to look at Evolution, a mail and calendar client, and Nautilus, a web browser and file manager, for more detailed 'historical' analysis. I tried two approaches: one used the normalized data, and identified periods where the signifier occurred more frequently with respect to everyday volume. The second approach used the actual signifier counts to see why that signifier occurred more frequently than other periods.

I looked at the normalized *Usability* events in Evolution, shown in Fig. 4.3. To eliminate bug reports and triaging events, I excluded these types of data from the query. Many bug reports are auto-generated, and contribute more noise than signal. For instance, one initial peak examined was related to the "Mass close of stale bugs $>=$ 4 months old." This generated a lot of noise as the signifiers in these reports are considered once more by the algorithm (since I treat any discussion on a bug similarly to mail threads).

With these events removed as noise, Fig. 4.3 shows a cluster of points in early 2000. Mailing list discussions at that time turned to a question about the default option for forwarding mail messages, e.g., ". . . I know this was discussed a few weeks ago . . . could it be implemented as an advanced option that has to be turned on and is off by default?" Later that year, in October, another spike in the graph can be attributed to a feature-freeze on Evolution and associated UI cleanups. As Evolution 1.4 is released in mid–2003, there is a small upward trend. Events at that time reflect problems with the new release, reflecting some UI changes. We still see some effects due to volume, such as the outlier near the end of 2003, where nearly one-third of mailing messages were usability related. The issue here is one of overall volume over the winter holidays. In this case a single mail thread about keyboard shortcuts consumed the discussions.

For my second approach, I used the actual signifier event counts, and targeted *Reliability* events for Nautilus. In November, 2000, 50 events occur. Inspecting the events, one can see that a number have to do with bug testing the second preview release that was released a few days prior. For example, one event mentions ways to verify reliability requirements using hourly builds: "As a result, you may encounter a number of **bugs** that have already been fixed. So, if you plan to submit **bug** reports, it's especially important to have a correct installation!". Secondly, in early 2004 there is a point with 29 events just prior to the release of Gnome 2.6. Discussion centres around the proper treatment of file types that respects reliability requirements. It is not clear whether these discussions are in response to the external pressure of the deadline or are just part of a general, if heated, discussion.

These investigations show that there is value to examining the historical record of a project in detail, beyond quantitative analysis. While some events are clearly responding to external pressures such as release deadlines, other events are often prompted by something as simple as participant interest, which seems to be central to the OSS development model. With respect to the nature of requirements evolution, I suggest that this is strong evidence that changes in requirements are in fact unanticipated: the maintenance approach, at least for these products, is reactive and not planned, suggesting a widespread understanding that one cannot anticipate requirements change.

Quality importance and project Recall that in the second question, I wanted to examine whether certain projects would be more concerned with software quality requirements than others. The importance of a requirement to a project is characterized by calculating the mean normalized occurrences of the signifier (such as *Usability*) over time. This controls for both project longevity and project size.

Table 4.9 lists the results; for space considerations, only three (representative) qualities are listed. I show the mean number of occurrences per week, normalized by dividing by the overall number of 'events' in that period, to eliminate the effect of volume. We would like to know, in other words, what proportion of all messages in that week were talking about the requirement of interest.

I used the extended signifier set (`ext`). Qualities cannot be inter-compared, because the signifier sets are not the same size. However, there is a difference among projects. I chose to focus on Nautilus and

Evolution, since both are projects of similar longevity, focused on file management and mail respectively. The *Efficiency* quality occurs in Nautilus discussions at a rate of 0.026 occurrences per week, and in Evolution at 0.012 occurrences per week – less than half as often. *Usability* is discussed 1.5 times as often in Nautilus, while other requirements, including *Portability*, show no difference. One possible explanation is that Evolution participants have a conceptual model of Efficiency that is a poorer fit to the signifier lists than the model Nautilus participants use. However, it does seem fair to conclude that projects have different interests with respect to software quality.

Threats to validity

- Construct validity – The main threat to construct validity is that the signifiers may omit relevant terms or phrases, e.g., "can't find the submit button" vs. "usability". Qualities are not directly comparable, since their respective signifier set sizes differ. *Usability*, for example, has 24 terms in its bubble, versus *Functionality* with 10. I conducted the error analysis to determine how accurate the bubbles are. Error validation should be conducted by more people to ensure inter-rater reliability. Many events are tricky to classify. Furthermore, I am assuming that projects share the ontology of software quality expressed in the quality model (ISO9126). A more domain-specific taxonomy would be useful. I relied on expert assessment to extract additional domain-specific terms, but these terms may not be a complete reflection of the words participating in the discussion of a specific quality.

- Internal validity – When I perform the regression analysis, assuming a linear relationship may not be a good model of the actual pattern these discussions follow. For example, the number of occurrences may be changing in response to some other variable, such as co-ordinated release dates, or by things such as developer illness. However, it is not clear from examining the data that other models would be more suitable – there is no evidence, from the experiments, that an exponential or log relationship exists, for example. I focused on the linear model as it is the simplest explanation of the pattern we would expect to see if quality discussions were increasing with time. Finally, source data may not capture all discussions regarding quality requirements – I omitted IRC chats, for instance. However, these data sources are most amenable to large-scale analysis. Follow-up with qualitative studies would be useful.

- External validity – The data originated from open-source projects, less than ten years old, from the Gnome ecosystem. Of these, the open-source nature of the project seems most problematic for external validity. Capra et al. [2008], for example, show a higher software quality in OSS projects than commercial projects. It would be interesting to determine whether a top-down directive to focus on software quality, or some other methodological change, would present as a noticeable spike on the event occurrence graph.

Models of quality requirements There is a rich history of discussion regarding software quality requirements, and quality models in particular. The main problem that arose in this study was that the quality models are (by design) very high-level. They provide a useful baseline from which to derive more specific models. However, it is useful to have a cross-product quality requirements model which can be used to compare software systems. Many questions are left unanswered when confronted by actual data: for instance, what is the relationship between product reliability and product functionality?

The challenge for researchers is to align software quality models, at the high level, with the product-specific requirements models developers and community participants work with, even if these models are implicit. One reason discussions of quality requirements were difficult to identify is that, without explicit models, these requirements are not properly considered or are applied haphazardly. We need to establish a mapping between the platonic ideal and the reality on the ground. This will allow researchers to compare maintenance strategies for product quality requirements across domains, to see whether strategies in, for example, Gnome, can be translated to KDE Linux Desktop, Apple, or Windows software.

Lessons Learned

This project used a novel analysis technique for conducting empirical research in Requirements Engineering. The technique was applied to study two specific questions concerning quality requirements. In accordance with Lehman's laws of software evolution, I hypothesized that there is growing interest in quality requirements within a developer community as a project matures. However, the analysis provides no evidence for this hypothesis. However, it is sometimes possible to use external events to explain patterns in the data. The presence of these external events provides evidence that it is unanticipated changes which dominate the tasks of software maintenance, and in this case, unanticipated changes in requirements (rather than, for example, corrective bug-fixes).

I then showed that there is a difference in how different projects treat software qualities – with some projects discussing certain quality requirements more than others. This may seem somewhat of a tautology; why would we expect otherwise? For one, this result highlights the difficulty in capturing notions of software "quality", the improvement of which is the subject of much of the maintenance that is conducted on software. For another, it serves to highlight for product owners the particular qualities which are the actual source of development effort (or at least, development discussion).

The ultimate goal is to be able to extract, from available sources, a list of requirements for a project, so that one can trace not just the 'physical' changes in the codebase, but also the evolving features and goals inherent in a project. This project focused on quality requirements, since they cut across projects. I was able to use data mining to extract, with reasonable accuracy, the occurrence of interest in particular qualities. With respect to the requirements problem, this project showed that the requirements problem is truly an on-going one. It is not sufficient to consider a single, absolute solution at a single point in time, because the underlying domain assumptions change, and drive changes in the software.

In the following section, I discuss a second project that tried to automate the process of labeling occurrences of software qualities. Among other techniques, it uses multi-label classifiers on more domain-specific taxonomies (e.g., database systems).

4.1.2 Applying Latent Dirichlet Allocation to Extract Requirements Topics

In this project, published as [Hindle et al., 2011], my co-authors and I took a more statistical approach to extracting requirements from corpora. The objective was to show that a topic modeling approach could both extract topics from the dataset, and then accurately label those topics, where appropriate, with the relevant software qualities that were the subject. In this section, I briefly highlight the results from this work. I introduce the idea of topics in Software Engineering, and explain the statistical techniques used. I then introduce the data sources we used. Following this, I explain the results we obtained, and conclude by explaining the relevance to requirements evolution.

Topics in Software Development

A key problem for practising software maintainers is gaining an understanding of *why* a system has evolved the way it has. This is different from *how* a system has evolved, which is well-characterized by the artifacts themselves. Looking back on streams of artifacts scattered across different repositories, inferring what activities were performed, when, and for what reasons, is hard to do without expert advice from the developers. If the objects of a 'why' question are reasons for system evolution, then the subjects of a 'why' question might be called development topics. These often are exhibited as latent topics that relate to many development artifacts such as version control revisions, bug reports, and mailing-list discussions. In this work we sought to provide a method of automatically labelling these development topics extracted from commit logs.

Topic modeling is a machine learning technique which creates multinomial distributions of words extracted from a text corpus. This technique infers the hidden structure of a corpus using posterior inference: the probability of the hidden structure given the data. Topic models are useful in software maintenance because they summarize the key concepts in a corpus, such as source code, commit comments, or mailing-list messages, by identifying statistically co-occurring words. Among other uses, it can quickly give developers an overview of where significant activity has occurred, and gives managers or maintainers a sense of project history.

In this project, my co-authors and I devised *automated labelled topic extraction.* The project addresses two gaps in the topic mining literature:

1. Topic mining of software has been limited to one project at a time. This is because traditional topic mining techniques are specific to a particular data-set. *Automated labelled topic extraction* allows for comparisons *between* projects.

2. Topic modeling creates word lists that require interpretation by the user to assign meaning. Like (1), this means that it is difficult to discuss results independent of the project context. This technique automatically, or with some initial training, assigns labels across projects.

The project expands on the work in §4.1.1 by automating the process of labeling events with appropriate topics, which in this case are once more ISO9126 Software Quality terms.

Data Sources And Methodology

Figure 4.5 gives an outline of the methodology. The project began by gathering source data and creating topic models. For semi-unsupervised labelling, we generated three sets of word-lists as signifiers for NFRs. In supervised learning, we trained data with manual annotations in order to match topics with NFRs. Finally, these topics were used to analyze the role of NFRs in software maintenance.

Generating the Data To evaluate our approach, we sought candidate systems that were mature projects and had openly accessible source control repositories. We selected systems from the same application domain, to control for differences in functional, rather than non-functional, requirements. We selected MySQL 3.23 and MaxDB 7.500 as they were open-source, partially commercial database systems. MaxDB started in the late 1970s as a research project, and was later acquired by SAP. As of version 7.500, released April 2007, the project has over $940,000$ lines of C source code[2]. The MySQL project started in 1994 and MySQL 3.23 was released in early 2001. MySQL contains $320,000$ lines of C and C++ source code. Choosing an older version of these projects allowed us to focus on projects which have moved further into the maintenance phase of the software lifecycle. We explicitly chose older

[2]generated using David A. Wheeler's *SLOCCount*, http://dwheeler.com/sloccount.

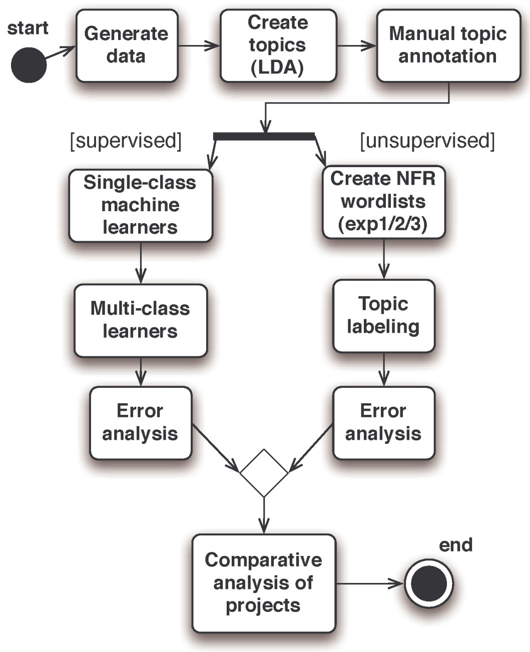


Figure 4.5: Research methodology process view.

versions of mature projects from a stable problem domain to increase the likelihood that there would be primarily maintenance activities in the studies.

For each project, we used source control commit comments, the messages that programmers write when they commit revisions to a source control repository. Most commits we observed had commit comments.

Commit comments are often studied by researchers, as they are the most readily accessible source of project interactions, and developers are often required to create them by the repository mechanism (e.g., Git). Additionally, relying only on commit comments makes our approach more generalizable, as we do not assume the presence of other artifact corpora. This is the most readily accessible source of project interactions for outside researchers, since developers often or always (e.g., Git) write such message. Our approach is more generalizable if it does not assume the presence of other corpora.

An example of a typical commit message, from MySQL, is: *"history annotate diffs bug fixed (if mysql-_real_connect() failed there were two pointers to malloc'ed strings, with memory corruption on free(), of course)"*. We extracted these messages and indexed them by creation time. We summarized each message as a word distribution minus stop-words such as "*the*" and "*at*".

For the commit message data-sets of each project, we created an XML file that separated commits into 30 day periods. We chose a period size of 30 days as it is smaller than the time between minor releases but large enough for there to be sufficient commits to analyze [Godfrey et al., 2009]. For each

30 day period of each project, we input the messages of that period into **Latent Dirichlet Allocation** (LDA), a topic analysis algorithm [Blei et al., 2003], and recorded the topics the algorithm extracted.

A topic analysis tool such as LDA will try to find N independent word distributions within the word distributions of all input messages. Linear combinations of these N word distributions are meant to represent and recreate the word distributions of any of the original messages. These N word distributions effectively form *topics*: cross-cutting collections of words relevant to one or more of our commit messages. LDA extracts topics in an unsupervised manner; the algorithm relies solely on the source data and word distributions of messages, with no human intervention.

In topic analysis a single document, such as a commit message, can be related to multiple topics. Representing documents as a mixture of topics maps well to source code repository commits, which often have more than one purpose [Godfrey et al., 2009]. For this paper, a topic represents a word distribution generated from a group of commit log comments which are related by their content.

We applied Blei's LDA implementation [Blei et al., 2003] against the word distributions of these commits, and generated lists of topics per period. We set the number of topics to generate to 20, because past experimentation showed that fewer topics might aggregate multiple unique topics while any more topics dilutes the results and creates indistinct topics [Godfrey et al., 2009].

We again used ISO9126 [Coallier, 2001] as the source for cross-project software quality labels. In order to validate our statistical results, we also created a validation corpus. For MySQL 3.23 and MaxDB 7.500, we annotated each extracted topic in each period with the six NFR labels listed above. Annotators did not annotate each other's annotations, but some brief inspection of annotations was used to confirm that the annotators were acting similarly. We looked at each period's topics, and assessed what the data — consisting of the frequency-weighted word lists and messages — suggested was the label for that topic. We were able to pinpoint the appropriate label using auxiliary information as well, such as the actual revisions and files that were related to the topic being annotated. For example, for the MaxDB topic consisting of a message "exit() only used in non NPTL LINUX Versions", we tagged that topic *portability*. Given the top-level annotations of *none*, *portability*, *efficiency*, *reliability*, *functionality*, *usability*, and *maintainability*, the annotators annotated each topic with the relevant label. Sometimes they used finer-grained annotations that would be aggregated up to one of these higher-level labels.

We validate classification performance using the Receiver Operating Characteristic area-under-curve value Fawcett [2006], abbreviated *ROC*, and the F-measure, which is the harmonic mean of precision and recall, i.e., $2 * (P * R)/(P + R)$.

ROC values provide a score reflecting how well a particular learner performed for the given data. ROC maps to the more familiar concepts of precision/sensitivity and recall/specificity: it plots the true positive rate (sensitivity) versus the false positive rate (1 - specificity). A perfect learner has a ROC value of 1.0, reflecting perfect recall and precision. A ROC result of 0.5 would be equivalent to a random learner (that is, issuing as many false positives as true positives). The ROC of a classifier is equivalent to the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance. We consider our labelling classifiers acceptable if they outperform a random classifier (0.5).

Semi-unsupervised Labels In this section I describe how to label topics based on dictionaries mined from sources external to the projects. I began by generating word-lists, which were synonyms for the ISO9126 qualities or NFRs. I created three word lists, of varying breadth, along the same lines as the word-lists in §4.1.1.

Using the three word-lists (hereinafter `exp1`, `exp2`, `exp3`), I labelled topics with an NFR where there was a match between a word in the list and the same word somewhere in the distribution of words that constitute the topic. A *named topic* is a topic with a matching NFR label. *Unnamed topics* occur where there is no such match, which may indicate either a lack of precision, or simply that this topic is not associated with non-functional requirements. All experiments were run on MaxDB 7.500 and MySQL 3.23 data. LDA extracted 20 topics per period for each project. This labelling is *semi-unsupervised* because the corpus is not derived from the project being analyzed, and I did not label the project's topics ourselves for a training set. The motivation behind this technique is that because most software often addresses similar issues, thus I can use the domain knowledge of software to label relevant topics.

Table 4.10 shows how many topics were labelled for MaxDB and MySQL.

Analysis – Detailed analysis is in [Hindle et al., 2011], but Figure 4.6 shows our ROC results for MaxDB and MySQL. *Reliability* and *usability* worked well for MaxDB in `exp2` and better in `exp3`. `exp1` performed poorly. MySQL had reasonable results within `exp2` for *reliability* and *efficiency*. MySQL's results for *efficiency* did not improve in `exp3` but other qualities such as *functionality* did improve. Many ROC scores were 0.6 or less, but our classifier still performed substantially better than random. It is important to distinguish what 'better' means here. ROC values seek to improve all measures. However, in some cases we may want to reduce the cost of false positives/negatives compared to the gain for true positives/negative. ROC values do not explain what is preferable in contexts in which the cost of a false positive is negligible, and that of a false negative is significant, or vice versa. In this case, we assumed that the relative weighting of these categories is equivalent. In some situations, for example, requirements elicitation, one may want to favour the recall and reduce precision.

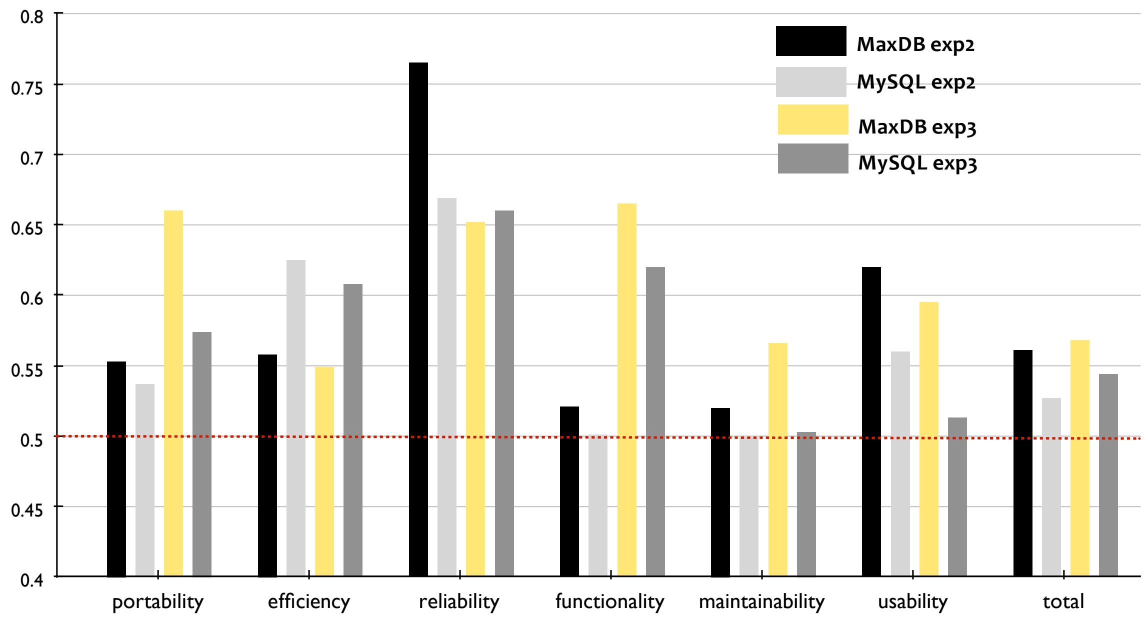


Figure 4.6: Performance, ROC values (range: 0–1), of semi-unsupervised topic labelling for each NFR and per word-list. The dashed line indicates the performance of a random classifier. This graph shows how well the semi-unsupervised topic labelling matched our manual annotations.

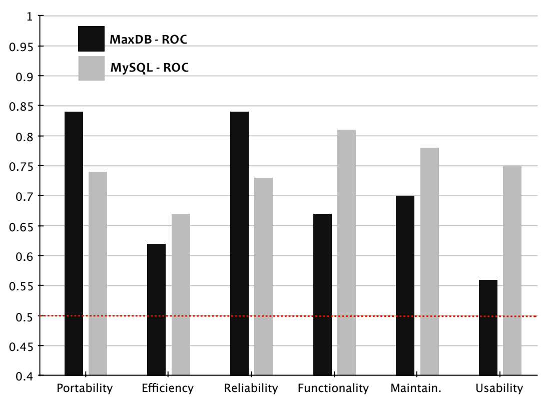


Figure 4.7: ROC value for the best learner per label for MaxDB and MySQL. Values range from 0–1. Dashed line indicates the performance of a random classifier.

Supervised Labels Supervised labelling requires expert analysis of the correct class/label to assign a label to a topic. In our approach, we use the top-level NFRs in the ISO9126 standard [Coallier, 2001] for our classes, but other taxonomies are also applicable.

We used a suite of supervised classifiers, WEKA [Hall et al., 2009], that includes machine learning tools such as support vector machines and Bayes-nets. The *features* we used are word counts/occurrence per topic; if a word occurs frequently enough in a topic we consider it a feature of the topic.

To assess the performance of the supervised learners, we did a 10-fold cross-validation [Kohavi, 1995], a common technique for evaluating machine learners. The original data is partitioned randomly into ten sub-samples. Each sample is used to test against a training set composed of the nine other samples. This is repeated nine more times, with each subsample used once as the training set.

Because our data-set was of word counts we expected Bayesian techniques, often used in spam filtering, to perform well. We tried other learners that WEKA [Hall et al., 2009] provides: rule learners, decision tree learners, vector space learners, and support vector machines. Figure 4.7 shows the performance of the best performing learner per label: the learner that had the highest ROC value for that label. The best learner is important because one uses a single learner per label. When applying our technique, for each NFR one should select the best learner possible.

Figure 4.7 shows that MaxDB and MySQL have quite different results, as the ROC values for *reliability* and *functionality* seem swapped between projects.

For both projects Bayesian techniques did the best out of a wide variety of machine learners tested. Our best learners, Discriminative Multinomial Naive Bayes, Naive Bayes and Multinomial Naive Bayes are all based on Bayes's theorem and all assume, naively, that the features supplied are independent. One beneficial aspect of this result is that it suggests we can have very fast training and classifying since

Naive Bayes can be calculated in $O(N)$ for N features.

The range of F-measures for MySQL was 0.21 to 0.77 with an average of 0.48, while MaxDB had a range of 0.17 to 0.61 with an average of 0.39.

The less-frequently occurring a label, the harder it is to get accurate results, due to the high noise level. Nevertheless, these results are better than our previous word-list results of `exp2` and `exp3`, because the ROC values are sufficiently higher in most cases (other than MaxDB *reliability* and MySQL *efficiency*). The limitation of the approach we took here is that we assume labels are independent; however, labels could be correlated with each other. Another approach we explored in the paper, but not reported here, is the use of multi-label learning; that is, use a learner that can associate more than one label per data point.

Applying LDA for Understanding Software Maintenance Activities

As I mentioned in the introduction, a key issue in software maintenance is understanding *why* a system has evolved the way it has. In this section we demonstrate the value of labelled topic extraction in addressing this issue. Labelled topics address *why* because they show the reasons for changes rather than *how*.

We investigated the history of the two large-scale database systems that we studied. We used our technique to show the topic of development efforts over time in each project. We motivated our investigation with two research questions:

1. *Do NFR frequencies change over time?* If a particular NFR was of more interest at one point in the life-cycle than another, this suggests that development activity shifted focus. For example, if a developer expected to see a recent focus on *reliability*, but instead *usability* dominated, they might re-prioritize upcoming work items.

2. *Do projects differ in their relative interest in NFRs?* A project manager, especially a systems-manager, would be interested in knowing whether a particular NFR, such as *reliability*, was more important for one project than another. This could be to confirm the initial design goals, or to track the progress on that quarter's objectives. The difference in NFR proportion is interesting because it implies a difference in focus between two projects.

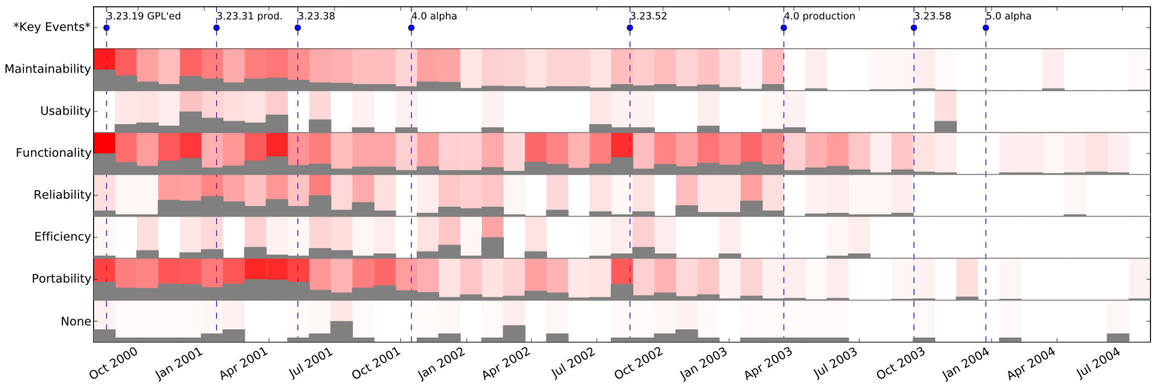
Figures 4.8a and 4.8b show the temporal patterns of NFR frequencies. There are two measures represented. One, the relative frequency, shown in the grey histogram boxes, represents the number of topics with that NFR in that period, relative to the maximum number of topics assigned to that NFR. For example, in Figure 4.8a we see a spike in *portability* and *functionality* frequency in September

2002. The second, absolute frequency, is shown using cell intensity, and compares the number of topics labelled with that NFR per period relative to the maximum number of labelled topics overall. For instance, Figure 4.8a shows that the NFRs *functionality*, *portability* and *maintainability* contain more labelled topics, since these NFRs have been more intensely shaded; one interesting stream is *efficiency* which shows periodic activity, does this suggest that once maybe efficiency related changes have longer lasting effects? The topmost row in each diagram lists historical events for that project (such as a release).

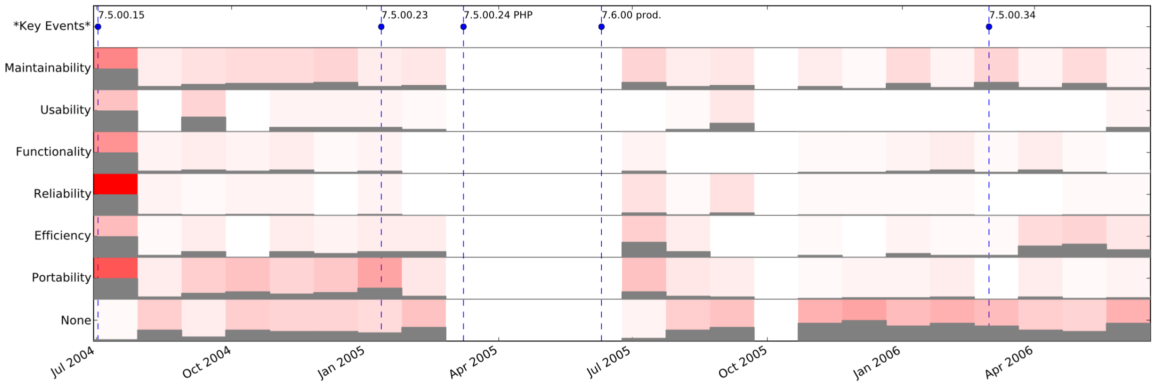
We analyzed each project's developer mailing list for external validation. We use *labelled topic extraction* to pick out the underlying NFR activity behind these events. For example, both projects show a high number of NFRs recognized at the first period of analysis. This is due to our window choice: we deliberately targeted our analysis to when both MySQL 3.23 and MaxDB 7.500 were first announced. For MaxDB, version 7.5.00 was released in December of 2003. We know that release 7.5.00.23 saw the development of PHP interfaces, possibly accounting for the simultaneous increase in the *portability* NFR at the same time. The gap in MaxDB (Figure 4.8b) is due to a shift in development focus (from February 2005 to June 2005) to MaxDB 7.6, which is released in June 2005.

The release of MySQL we study (Figure 4.8a) was the first to be licenced under the GPL. Version 3.23.31 (January, 2001) was the production release (non-beta), and we see a flurry of topics labelled with *functionality* and *maintainability.* After this point, this version enters the maintenance phase of its lifecycle. In May 2001, there is an increase in the number of topics labelled with *portability.* This might be related to release 3.23.38, which focused on Windows compatibility. Similarly, in August, 2002, both *functionality* and *portability* are frequent, and mailing list data suggests this is related to the release of version 3.23.52, a general bug fix with a focus on security (a component of the *functionality* NFR in the ISO9126 model). After this point, efforts shift to the newer releases (4.0, 4.1, 5.0). We now address our research questions, which are restatements of the hypotheses presented in §4.1.1:

1. Do NFR frequencies change over time? — In both projects the frequencies generally decreased with age. However, there are variations within our NFR labels. In MySQL, *usability* and *efficiency* do not appear very often in topics. A proportionately smaller number of commits addressed these NFRs. Certain peaks in topic numbers coincide with a particular emphasis from the development team on issues such as new releases or bug fixes. This suggests that maintenance activity is not necessarily strictly decreasing with time, but rather episodic and responsive to outside stimuli. This supports the notion that unanticipated requirements change is a main driver of software maintenance activity. While one might expect this to be the case, theories of software evolution have suggested that maintenance must be performed regularly as software ages (much like a maintenance schedule for cars).



(a) MySQL 3.23



(b) MaxDB 7.500

Figure 4.8: NFR label per period. Each cell represents a 30-day period. Grid cell intensity (saturation) is mapped to label frequency relative to the largest label count of *all* NFRs. Grey histogram bars indicate label frequency relative to that particular NFR's largest label count. Dashed vertical lines relate a project milestone (**Key events**) to our topic windows.

In MaxDB, we can observe that *Maintainability* topics became more prevalent as MaxDB matures. This is likely due to our analysis time-frame for MaxDB being shorter than the time-frame for the MySQL product.

2. Do projects differ in their relative topic interest? — Yes. MySQL 3.23 had proportionally more topics labelled *functionality*, while MaxDB had proportionally more *efficiency* related topics. MaxDB was a very mature release "donated" to the open-source community, whereas MySQL was in its relative infancy, and security problems were more common (security is a component of *functionality* in the ISO9126 model). In both cases *portability* was a constant maintenance concern and was prevalent throughout the lifetime of the projects. It may surprise developers how often portability arises as a concern.

Implications for Requirements Evolution

The preceding techniques have allowed us to analyze the time-series behaviour of quality requirements, and as such, give us some insight into how these qualities are evolving over time.

While an unsupervised technique such as LDA is appealing in its lack of human intervention, and thus lower effort, supervised learners have the advantage of domain knowledge, which typically means improved results. Creating annotated topics (i.e., manual labels) for training is painstaking, but with a suitably representative set of topics, the effort is acceptable. To annotate *all* topics took us approximately 20 hours per project, but we estimated only 10% of the topics need annotation to produce useful results. This is based on the 10x Crossfold analysis we performed: each 10% slice performed equally well.

Since this work focuses on labelling natural language commit log comments I feel it can be adapted to other natural language artifacts such as mailing-list discussions and bug reports, even though that was not evaluated in this project. Bug reports might not exhibit the same behaviour as commits in terms of dominant topics.

Threats to Validity

Construct validity – we used only commit messages rather than mail or bug tracker messages. To extend further we would need matching repositories for each project. Possibly they would have influenced our results, but there would be a degree of correlation between the corpora.

Our taxonomy for software NFRs is subject to dispute, but seems generally accepted. Finally, there are exogenous sources, such as in-person discussions, which we did not access, an omission as noted in Aranda and Venolia [2009].

Internal validity – We improved internal validity by trying to correlate and explain the behaviours observed in the analysis with the historical records of the projects. We did not attempt to match our results to any particular model. We were unable to assess inter-rater reliability.

External validity – Our data originated from OSS database projects and thus might not be applicable to commercially developed software or other domains. Furthermore, our analysis techniques rely on a project's use of meaningful commit messages, although we feel this is the most frequently occurring form of developer comments.

Reliability – each annotator followed the same protocol and used the same annotations. However, only two annotators were used; their annotations could be biased as we did not analyze for inter-rater reliability because they did not rate the same documents.

This study showed the feasibility of labelling development topics automatically using machine learning techniques. The error rates are still on the high side, which means that some topics are going unlabelled when they should be labelled, and vice-versa. Nonetheless, LDA analysis combined with the cross-project taxonomy provided some important insight into how projects are managing non-functional requirements. In particular, we can see how the level of effort dedicated to these requirements has changed over time, an important consideration if we are to design tool support for evolving requirements.

4.1.3 Related Work in Mining Requirements

Alongside the explosion in open-source software development that has occurred since the early 1990s has developed a research capability hitherto relatively unused: repository mining. This research area uses the publicly available repositories from open-source projects to produce empirically-validated work in software engineering. Tu and Godfrey [2002] were among the first to use this data to assess software evolution. They discussed how well the growth of Linux was predicted by laws that Lehman and Belady [1985] proposed. More recently, German et al. [2007] predicted OSS growth using time series analysis. They also refer to release windows as a useful unit of analysis.

Mens et al. [2008] conducted an empirical study of Eclipse, the OSS development environment, to verify the claims of Lehman et al. [1997]. They concerned themselves with source code only, and found Law Seven, "Declining Quality", to be too difficult to assess: "[we lacked an] appropriate measurement of the evolving quality of the system as perceived by the users [Mens et al., 2008, p. 388]". This paper examines the notions of quality in terms of a consistent ontology, as Mens et al. call for in their conclusions.

Two relevant sub-disciplines are the mining of requirements from repositories, and the use of topic extraction to generate latent development subjects.

Mining Requirements from Repositories

The idea of mining requirements from historical artifacts was first captured by the work on narratives of software systems shown in [Antón and Potts, 2001]. The authors looked at a large telephony software system from the user perspective, and analyzed the continuing change in features the system exhibited.

Most closely related to the work in my thesis is the studies of Jane Cleland-Huang and her colleagues. They published work on mining requirements documents for non-functional requirements (quality requirements) in [Cleland-Huang et al., 2006]. One approach they tried was similar to the one I used, with keywords mined from NFR catalogues found in [Chung et al., 1999]. They managed recall of 80% with precision of 57% for the Security NFR, but could not find a reliable source of keywords for other NFRs. Instead, they developed a supervised classifier by using human experts to identify an NFR training set. There are several reasons I did not follow this route. One, I believe my study had a more comprehensive set of terms due to the taxonomy chosen.

Secondly, I wanted to compare across projects. Their technique was not compared across different projects and the applicability of the training set to different corpora is unclear. A common taxonomy allows me to make inter-project comparison (subject to the assumption that all projects conceive of these terms in the same way). Thirdly, while the objective of Cleland-Huang's study was to identify new NFRs (for system development) my study assumes these NFRs are latent in the textual documents of the project. Finally, the source text used is less structured than their requirements documents.

[Massey, 2002; Scacchi, 2002; Scacchi et al., 2005] have been three studies that looked at the topic of requirements in open-source software. Their work discusses the source of the requirements and how they are used in the development process. German [2003] looked at GNOME specifically, and listed several sources for requirements: leader interest, mimicry, brainstorming, and prototypes. None of this work addressed quality requirements in OSS, nor did it examine requirements trends.

The difference between projects in level of interest in particular quality requirements was detailed in several case studies described in Doerr et al. [2005], which describes a methodology which starts with ISO9126 and 'tailors' the requirements analysis to specific NFRs.

Holt et al. [2007] examined release patterns in OSS. They showed that there is a difference between projects regarding maintenance techniques. This supports our result that software qualities are not discussed with the same frequency across projects.

Extracting requirements from source code was attempted in [Yu et al., 2005b] and [Tun et al., 2008]. Yu et al. [2005b] used code call graphs and the refactoring technique 'extract method' to create a goal model from code. The technique was largely manual, as it required an analyst to examine the various methods and abstract them into a goal or task. These were then connected as a statechart using the call graphs.

In [Tun et al., 2008] the authors extract Jackson-style problem frames from source code. The approach builds on [Yu et al., 2005b] by relying on feature descriptions to create the problem structure.

Most of this related research examined project source code. Software mining of other project corpora is less common. Similarities exist with approaches that begin with structured taxonomies, as with the Hismo software ontology [Girba and Ducasse, 2006]. Rigby and Hassan [2007] used a general purpose taxonomy to classify developer email according to temperament.

Related Work in Topic Mining

Topic mining of software repositories starts with Mockus and Votta [2000], who studied a large-scale industrial change-tracking system. They used WordNet for word roots as they felt the synonyms would be non-specific and cause errors. Mockus et al. validated their labels with system developers. Since in this work we studied multiple projects, instead of a single project, these interviews are infeasible (particularly in the distributed world of open-source software).

Marcus et al. [2004] use Latent Semantic Indexing (LSI) to identify commonly occurring concerns for software maintenance. The concerns are given by the user, and LSI is used to retrieve them from a corpus. Unlike LSI, topic modelling generates topics that are independent of a user query, and relate only to word frequencies in the corpus. Some results were interesting, but their precision was quite low.

With ConcernLines, Treude and Storey [2009] show tag occurrence using colour and intensity. They mine developer created tags, for example 'milestone 3', and used these in order to analyze the evolution of a single product. The presence of a well-maintained set of tags is obviously essential to the success of this technique.

For Baldi et al. [2008], topics are named manually: human experts read the highest-frequency members of a topic and assign a label accordingly. As discussed earlier, given the topic *"listener change remove add fire"*, Baldi et al. would assign the label *event-handling*. The labels are reasonable, but still require an expert in the field to determine them. Furthermore, these labels are project-specific, because they are generated from the data of that project. E.g., we might have a label *Oracle* in the MySQL case, since Oracle owns MySQL.

Mei et al. [2007] use context information to automatically name topics. They describe probabilistic labelling, using the frequency distribution of words in a topic to create a meaningful phrase. They do not use external domain-specific information as we did, but we do not generate phrases from the topics.

Godfrey et al. [2009] proposed a windowed method of topic analysis that this project extended with labelled topics, NFRs and new visualizations.

This section has presented our work on mining software repositories in order to gather data on how requirements, in the form of software quality requirements, are evolving. In the next section, I look at a case study that uses design objectives to capture a broader sense of requirements evolution, one that is not restricted to quality requirements.

4.2 Tracing Software Evolution History with Design Goals

When designing software for evolvability, it is important to understand which particular designs have worked in the past – and which have not. This section argues that understanding the *history* of how a requirements problem was solved is valuable in setting the context for finding future solutions. In the language of §2.1.2, such a history is a way of reconstructing the form of the requirements problem at a particular point in time, along with the specific updates to the specification that were proposed as a result of changes to that requirements problem.

There is no formal discipline of software history. While there is an active body of research in information technology (IT) and innovation management, which seeks to understand how to maximize value from IT spending, this research often ignores the meaningful technological underpinnings of such tools. I suggest that the study of design history should be extended to software artifacts. Notions of technical and social context explain how, and why, certain solutions evolved as they did. I apply these concepts to the history of distributed computing protocols. I conclude with observations drawn from this history that suggest designing software for evolvability must consider the history of similar applications in the requirements analysis.

4.2.1 Motivating the Study of Software History

Lehman and Ramil [2003] describe the *nounal* view of software evolution as being concerned with "the nature of evolution, its causes . . . [and] its impacts . . . (p. 35)." One way of looking at software evolution is to examine the low-level artifacts, such as metrics about lines of code, number of modules, or commit logs. What is missing in such studies is a broader look at the the causes of software evolution. Such an examination has two aspects. One, higher-level artifacts need to be considered. Software requirements fulfil this objective. They are high-level expressions of intentions, describing a need in the environment that can be met by instantiation in software. The second aspect is that such a study must be historical, looking at multiple products over a broad sweep of time.

What form should such a study take? One form might be a broad-brush history. For example, Mahoney [2004] looks at the entire field of software engineering. Most histories of this type, including many found in the IEEE journal *Annals of the History of Computing*, focus on software engineering in the large, examining the people involved, the decisions made, and the end results. From a software evolution perspective, though, the interesting questions are not "what happened", but rather, 'why did design X get chosen and not design Y'?

The motivation for this section is to demonstrate that an enquiry into the goals a particular software technology tried to satisfy – its high-level requirements – is very helpful to maintainers and implementers of today's new technologies. Studying the change in how technology innovators understand a particular requirements problem captures several things. One is an understanding of the large-scale domain assumptions for a particular technological problem. The second is a series of lessons in problems faced and challenges overcome, that is, the way the requirements evolution problem was solved. Such problems and challenges may well repeat themselves in the future.

The argument is laid out as follows. First, I introduce other work on this topic, and highlight differences. I then introduce the case study used to argue for conducting disciplined software history. The case study is a narrative history of distributed computing protocols. This section concludes with my interpretation of the evolution of this technology. I then suggest some lessons the evolution of these requirements provides for better understanding the requirements evolution problem.

4.2.2 Related Work in Software Histories

Antón and Potts [2001] report on a feature-oriented analysis of corporate telephony offerings, a narrative they term functional morphology: the shape of benefits and burdens of a system. They excavate features from past iterations of the system, and use that to argue for their interpretation of the changes in requirements. The principle difference with this section is their focus on features rather than requirements. They worked with closed-source software which prevented them from delving into the requirements. Similarly, Gall et al. [1997] describe how they used software releases to gain insight into evolutionary trends. From a product release database, they examine the growth-rate and change in system modules. There is no focus on why changes are made. Spira [2003], in the short-lived publication *Iterations*, gave a brief history of the TCP/IP protocol and its creation. This was, again, a history that focused on "what happened" rather than why. We are interested in how changes in domain assumptions and requirements drove the "what".

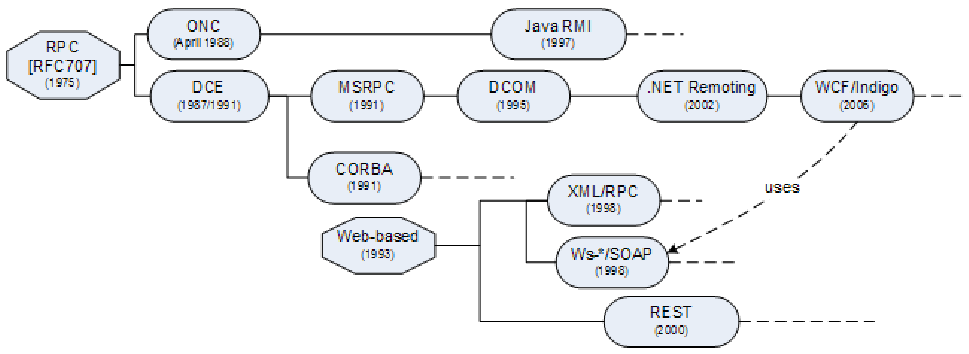


Figure 4.9: Phylogeny of distributed computing protocols

4.2.3 A history of distributed computing protocols

To demonstrate the usefulness of an intentional history of software evolution, I present a case study on distributed computing protocols. What I looked for is the design decisions, rationale and requirements that motivated each particular implementation or proposal. We can then compare those decisions and rationale with preceding and subsequent work, trying to construct the narrative for the work on distributed computing. Figure 4.9 shows a rudimentary ‘phylogenetic tree’ for this history.

Telling disciplined software history

The aim of disciplined history is a plausible explanation of past events [Dixon and Alexander, 2006]. Plausible explanations use sources (evidence) to build an argument about why events occurred. Historians traditionally have used primary sources such as letters, diaries, and newspapers from the past. For telling the story of a software technology, that is, its evolutionary record, the primary sources such artifacts as code, mailing lists, and release-specific documentation, such as READMEs. From these sources, a historian constructs a narrative, providing facts and interpretation.

For this look at the intentionality behind specific technologies, the primary artifacts are the published specifications and proposals for each protocol. These capture, in the words of the innovators, the reasons behind the new proposal. I also draw on histories of the software industry at specific dates to reconstruct the wider context behind each decision. Finally, I draw on the details in the specifications to establish the *de facto* intentions of the protocol. One gap in the reconstruction is a consideration of how the specific protocol evolved – for example, how CORBA changed from version 1.0 to version 2.0. I leave these detailed analyses to Chapter 5. I also omit analysis of the actual implementations of the protocols, in favour of a look at the intentions, rather than the reality. Needless to say, how well a specification is implemented has a strong effect on future changes to it and competing specifications.

Definition of Distributed Computing

Computer programs consist of data and definitions for operations thereon. The operations are typically defined as procedures, functions, or methods, and define the steps – the algorithm – for accomplishing the desired output state. Programs can also be seen as resource manipulators - from a start state to an end state. *Distributed computing* is the ability to manipulate resources on disconnected, heterogeneous systems. Such capabilities were one of the reasons computers were networked in the first place. Particularly in the early days of computing, resources like bandwidth or processing power were scarce.

If one is to distribute computation – for example, an algorithm for calculating taxes owed – there are many questions that must be answered. Are integers big-endian or little-endian? Are we using an ASCII character set? How does System A acknowledge receipt of System B's request? These questions are answered by creating a protocol. Protocols are standardized descriptions of resources and processes designed to maximize interoperability.

This section describes the protocols that have been proposed over the years for doing distributed computing. For each protocol, the original descriptions of the protocol were used to reconstruct the high-level goals the particular protocol was intended to address.

Early Years

The development of the ARPAnet in the late 1960s was the start of the networking age. A few years later, local- and wide-area networking standards, such as IBM's SNA and Ethernet, made it possible for co-located machines to communicate. Operating systems had earlier developed the notion of inter-process communication, where threads could exchange data, and such notions paved the way for distributed computing.

The initial impetus for a distributed computing protocol can be traced to such programs as FTP and Telnet. Telnet allowed users to run programs on another system [Postel and Reynolds, 1983]. In 1975, recognizing that a number of these ARPA-based processes duplicated functionality, James White proposed the *remote procedure call* [White, 1975]. What White argued for was the ability for machines to do the same thing as humans did with Telnet. He critiqued Telnet, and related programs, for missing out on higher-order functionality:

> "The syntax and semantics of these interchanges, however, vary from one system to another and are unregulated by the protocol; the user and server processes simply shuttle characters between the human user and the target system" White [1975, p.3].

Before his proposal, machine-to-machine communication had to be handled at a low-level, with the communication functions being written anew.

White listed eight requirements for a replacement protocol:

- Allow arbitrary named commands to be invoked on the remote process;
- Report on the outcome of such an invocation;
- Permit arbitrary numbers of parameters;
- Represent a variety of types;
- Allow for the process to return results;
- Eliminate the user/server distinction;
- Allow for concurrent execution;
- Suppress any response if asked.

White's list of requirements give some insight into the problems he was trying to solve. For example, he wanted to remove any constraints on remote method invocation, to make remote procedures 'feel' like local calls. His proposed type system was designed to deal with heterogeneity, so that remote systems would be able to talk to a local machine in the same syntax. He also makes mention of the notions of synchronicity and blocking, i.e., what to do with long-running or complex calls. Finally, his proposal emphasizes the need for a "simple, rigidly specified command language". This goal would eliminate the profusion of heterogeneous programming languages which he saw as an obstacle to further progress in distributed computing. All of these requirements would come to inform distributed computing protocols for the next 15 years.

RPC - Remote Procedure Call

The first substantial work on creating an implementable standard for distributed computing was that of Birrell and Nelson [1984] in 1984, working on behalf of Xerox. The Xerox proposal drew on White's work with the following different emphases. Like White, they highlighted the requirements that the proposal make distributed resource sharing indistinguishable from local method calls, down to emulating program semantics - i.e., "the semantics of remote procedure calls should be as close as possible to those of local (single-machine) procedure calls [Birrell and Nelson, 1984, p. 43]". They focused on 'clean and simple semantics'. Complex semantics would make debugging more difficult, and development tools like IDEs and distributed debuggers were in their infancy. Efficiency and latency were also concerns, given the slower network speeds of the day. Finally, the proposal emphasizes generality in the interface, allowing, like White, arbitrary commands and parameters. The goal was to allow the protocol to be as flexible as necessary.

Language constructs dictated elements of the distributed computing paradigm: the use of the Mesa language – a modular, imperative language – suggested the use of a remote procedure call paradigm over a message-passing one, as noted in the protocol. As well, exception signals were generated that would appear indistinguishable in Mesa from local exceptions. The architecture chosen for the RPC proposal was that of 'stubs and skeletons'. A remote procedure would generate a stub for the local program to call. The RPC layer would handle procedure calls to the stub, marshaling and serializing them over the network, where the skeleton would hand off to the remote procedure. The local stub served as the interface description, a contract the local caller could rely on. The stub and skeleton mechanism is fairly rigid in terms of evolvability; for any improvement on the remote site, an entire new stub needs to be generated locally.

ONC and DCE Following the Birrell and Nelson paper in 1984, several software companies began to develop RPC implementations. The software industry was undergoing a tremendous growth period [Steinmueller, 1995]. Two standards enjoyed primacy. These were DCE and Open Network Connectivity [Sun Microsystems Inc., 1988], proposed by Sun Microsystems in 1988, and the basis for their still-popular Network File Sharing (NFS). While following most of the requirements listed in [Birrell and Nelson, 1984], Sun aimed for transport independence, so programs could be reused to some extent. They explicitly added state to the remote procedure by adding a transaction ID that served to identify each call uniquely. Overhead was also important, so the protocol ignores reliability, allowing the transport to handle that. There was also a mention of authentication, perhaps the first realization that distributed procedures created several security risks. Finally, the specification gave a detailed definition of call semantics – such as at-most-once, at-least-once (execute the client's invocation at most once), and idempotent (no changes no matter how many calls).

While Sun's ONC was gaining traction, other vendors and institutions, such as Cambridge, HP, and MIT, developed RPC implementations according to their interpretation of the issues. Eventually this gave rise to a standardization process, under the aegis of the Open Software Foundation, called DCE, the Distributed Computing Environment [Johnson, 1991], in 1991. The larger context was the work of Sun and AT&T on Unix, which threatened to dominate the efforts of the other vendors. For example, MIT wanted an RPC that worked with UNIX, and to which they could backport existing applications.

Like earlier efforts, DCE was designed with heterogeneous, distributed networks in mind. Unlike

Sun's effort, DCE was explicitly designed to work across multiple operating systems, since by this time, there actually were multiple operating systems, unlike the assumptions that White could make. DCE was more than just RPC, however. In order to compete with NFS, the DCE proposal included a distributed file system, time server, and concurrent programming support. New design requirements were added as well. For the first time, a distributed computing standard recognized the notion of service-orientation, an architectural style that separates business rules from application logic. DCE provided rudimentary support for this by recognizing that a procedure's invoker may not necessarily be identical to its consumer. DCE also explicitly supported bindings for different languages, a recognition of the organizational structure of OSF.

CORBA

Perhaps one of the most influential distributed computing protocols, or rather, suite of protocols, was CORBA. CORBA is an object-based distributed computing specification. Today it exists in a largely legacy software context. Rather than remote procedures, self-contained components are transferred by middleware mediators called Object Request Brokers. A product of an industry standards group called the Object Management Group (OMG; also responsible for UML), the CORBA specification was first proposed in 1991, but this standard release is generally thought to be of poor quality; for example, it did not define how to communicate between different object brokers [Chappell, 1998]. By late 1996, major inconsistencies were cleared up and the second version released [Boldt, 1995; Vinoski, 1997].

CORBA was an enormous suite of standards with widespread industry involvement. A key goal, common to RPC technologies as well, was the separation of interface from implementation. Although such encapsulation was done in RPC as well, the wider context was the rising use of the object-oriented programming paradigm [Steinmueller, 1995]. Key to this concept was the Interface Definition Language (IDL). The IDL provides language-independent declarations that are then mapped into a programming language to create the object stubs. Clients invoked only those operations that a remote object exposed through its IDL interface. A second innovation was the introduction of multiple distributed systems. The ORB one used could also communicate with other ORBs through a protocol called the Internet InterORB Protocol (IIOP). IIOP defines the protocol by which messages are passed. Finally, CORBA made reference to scaling requirements, e.g., load balancing. Previous efforts had not made this issue as high-priority as the CORBA standard did.

CORBA was quite successful for its vendors, yet has lost mindshare since the late 1990s. Vendors involved in the process were mainly interested in developing proprietary products, which meant the standardization process suffered as they focused on their own products [Henning, 2006]; CORBA, like

DCE, mapped its standards to programming languages, but produced no mappings for Java until that language had developed its own tools (J2EE). Similarly, the XML revolution (see below) left CORBA behind. In the context of the requirements problem, CORBA suffered from an overly large solution space, and a lack of focus on important software qualities.

To emphasize that not all reasons are technical, there was a business reason for CORBA's troubles. The industry context at the time was the dominant position of Microsoft, and its technologies. Competing against Microsoft placed the CORBA specifications in a challenging position. Many IT managers were hesitant to abandon Microsoft, yet appreciated the cross-platform abilities of CORBA.

Java: RMI and EJBs

In late 1995 Sun Microsystems released Java 1.0. The Java platform soon nurtured a large community of developers. The initial support for distributed computing in the Java platform came from the bundled `java.rmi` library. RMI stands for Remote Method Invocation, and as the name suggests, is an RPC-like remote object protocol [Sun Microsystems Inc., 1999]. The real power of RMI came from the way it tightly integrated the distributed object model into the Java programming language. The other important features of RMI, other than its excellent integration, are its simplicity and efficiency (unlike CORBA). One reason for this simplicity was the assumption that people would only develop in a single language, which would compile to bytecode rather than a particular operating system.

By 1998 developers of complex distributed systems were running into difficulty. As applications grew in size, the facilities in existing architectures, like CORBA or RMI, seemed inadequate for handling complexity. Sun's solution was the release of the Enterprise JavaBeans (EJB) platform [Matena and Hapner, 1999]. This release heralded the development of application servers, where a central server delivers applications to client machines. The central server contains the application and business logic, which addresses (some of) the issues of complexity. Contemporaneously, Sun and Oracle were developing the notion of the network computer.

The EJB specification emphasized three benefits: 1) an underlying framework, initially using CORBA middleware, that abstracted issues related to transport; 2) separation of business logic from application logic, such as user-facing interfaces; and 3) portability and reusability from a component model. Again, these were backed by a common programming language and the promise of “write-once, run anywhere”. These were not new ideas, but again, were well-integrated for a large, pre-existing market. By this point, as listed in the specification, Sun was beginning to see the value in ‘web services’. The underlying transport mechanisms, such as IIOP, were less important that the abstraction the specification afforded.

Microsoft: DCOM

Microsoft's competing technology for CORBA, Java, and RPC was an extension of the DCE-RPC standard they called DCOM, for Distributed Component Object Model. DCOM saw widespread adoption due to its tight integration with the large installed base of Microsoft operating systems and server tools. The DCOM specs [Microsoft Corporation, 1996; Horstmann and Kirtland, 1997] mention several design concerns, including the importance of garbage collection and memory management; security via access control lists; ease of deployment; and the ability to version remote objects: "With COM and DCOM, clients can dynamically query the functionality of the component." Again, like Sun's RMI and EJB proposals, there was a set of simplifying assumptions that promised much greater ease of development.

The 1990s ended with a focus on the notion of SOA, or service-oriented architecture. This was the idea of separating business logic from application logic. and establishing service contracts via interface definitions. SOA systems were loosely coupled and focused on the notion of reusability and encapsulation. The next decade, up until the present day, has seen this mindset gain primacy.

The Web Years

In 1990, Tim Berners-Lee proposed the World Wide Web [Berners-Lee and Cailliau, 1990], based on what would become Internet standards: HTTP for interactions and URIs for resource identification. Distributed computing technologies leveraged these new opportunities to develop new protocols and technologies. These opportunities came in the form of relaxed domain constraints, such as all computers soon having a Web browser, and that HTTP-based servers were ubiquitous. A contemporaneous standard, XML, allowed for a ubiquitous and extensible format for data exchange. "What distinguishes the modern Web from other middleware is the way in which it uses HTTP as a network-based Application Programming Interface (API) [Fielding, 2000, §6.5.1]." This allowed protocols to abstract away the network requirements and focus on the semantics of the distributed services.

Web Services As XML was being standardized, there was ongoing work on SOAP, the Simple Object Access Protocol. Due to delays, and in the interests of creating a usable, light-weight RPC mechanism free from the influence of Microsoft, XML-RPC Winer [1999] was released as a specification. XML-RPC used XML, which permitted introspection, was extremely simple (2 pages), and quickly gained support. The data representations chosen were fairly primitive, however, and better support for data types was needed. This is another example of a standard that could make simplifying assumptions due to vast improvements in the underlying technologies. XML, for example, removed the need to define a serialization format.

The SOAP standard [Box et al., 2000] (Simple Object Access Protocol) was driven by Microsoft as a way to address the growing influence of Sun's EJB activities. Up to that point, Microsoft's distributed computing solutions (DCOM) were Microsoft-specific. SOAP was a means to communicate with any other heterogeneous computing platforms, as long as that platform could parse XML. This was motivated by the need to work with legacy systems, such as old mainframes. One of SOAP's design goals was flexibility. As one of the designers mentions, "we looked at the existing serialization formats ... and RPC protocols ... and tried to hit the sweet spot that would satisfy the 80% case elegantly but could be bent to adapt to the remaining 20% case."[3]. To this end, a number of features common to, for example, DCOM, were omitted, such as garbage collection. To define interfaces – what endpoints expect to receive as a message, and can return as a result, WSDL was introduced Christensen et al. [2001]. SOAP also took advantage of existing work on XML datatypes, in the form of XML Schema.

The design goal for SOAP was to be fairly neutral architecturally. SOAP can be used in a number of different distributed computing architectures, such as message passing and remote procedure calls: "SOAP is fundamentally a stateless, one-way message exchange paradigm [Box et al., 2000]."

To define some more complex features on top of SOAP, such as transaction support, reliability, security, and so on, the WS-* process [Booth et al., 2004] defined more standards. Web services are SOA-style interactions over the web, usually using SOAP and WSDL. The consensus to date seems to be that this heavyweight approach has largely been superseded by the REST approach (below).

REST An alternate approach to SOAP's message passing paradigm is REST, REpresentational State Transfer. Defined in 2000 in Fielding [2000], REST is "a small set of simple and ubiquitous interfaces to all participating software agents. Only generic semantics are encoded at the interfaces. The interfaces should be universally available for all providers and consumers." The idea is to move away from encoding the operations in an object, and use the fairly universal HTTP actions: PUT/GET/DELETE/POST, among others. This supports generality, and has the benefit that the effects of each action are universally understood; e.g., GET is an idempotent operation (according to the HTTP specification). The complexity in the interaction is handled by the resource state: "[unlike SOA], for a client to invoke a REST service, it must understand only that service's specific data contract: the interface contract is uniform for all services." [Vinoski, 2007]. Fielding's thesis mentions several design goals for REST, induced by its stateless nature. These are visibility: the entire nature of a request (invocation) is embedded in the message; scalability on the server, as all state is client-side; and reliability, because partial failures can be dealt with.

[3]http://webservices.xml.com/pub/a/ws/2001/04/04/soap.html

REST, as of June, 2007, was the protocol with the most traction among web developers; for example, Amazon's web services available in the REST style are used 85% of the time, and the SOAP interfaces 15% of the time[4]. REST is sufficiently simple, and there are good examples of how well it scales – since HTTP is a REST-style architecture, we can assume REST applications will scale as well as the Web does. There are concerns about security, but the advantage of REST – and its fundamental difference with the other technologies we have looked at – is that it completely abjures the interface contract and transport mechanisms. These are managed by the underlying protocol, typically HTTP. Security, for example, can be handled by the HTTP-Authentication standards. REST's only interest is in the transfer of application state.

REST concludes the survey of distributed computing technologies. I have shown how, for each technology, the designers focused on particular requirements in order to address challenges they identified. In the following section, I present my interpretation of this narrative as it relates to the wider goals of managing software evolution.

4.2.4 Interpretation

This section takes the specific details from each of these distributed computing technologies and ties them together in a coherent narrative. What is common to all such distributed computing models is the primary distributed computing requirements set, identified in `RFC707` and unchanged since. This is because the underlying challenges and opportunities remain similar. The ability to access remote services and to abstract implementation details are compelling. Where there are differences, they occur closer to the implementation level.

What we see, then, is not major revisions in this feature set - all technologies, for example, abstract implementation details of the remote service – but incremental updates as new problems with existing implementations are identified. And as the context changes – for example, as the World Wide Web, HTTP, and URLs gain massive and widespread acceptance – the protocols evolve to leverage that context. With Birrell and Nelson [1984], the concept of remote procedures was shown to be feasible. However, once implemented, certain challenges arose. These included security, scaling, and transparency. With the rise of OOP, and the industry dominance of certain players such as Microsoft and Sun, pressure grew for smaller vendors to embrace cross-vendor standards to compete. These forces led to the development of CORBA, DCOM, and RMI. The limits to these technologies became more apparent as systems grew in size. That by the mid-1990s most servers used common transport standards, namely HTTP, and

[4]http://www.oreillynet.com/pub/wlg/3005

common interchange format (XML), supported the development of XML-based standards like SOAP and XML-RPC. Today we are witnessing the evolutionary competition of the web services standards with the goals of REST, that scale, generality and abstraction are best accomplished by focusing solely on resources identified by URIs.

The following are some of the lessons suggested by this study for designing for software evolvability (in the large):

1. Vendor lock-in is a blessing and a curse. In the case of DCOM, tight integration with the dominant platform of the day made development easier. At the same time, extending applications beyond that platform was essentially impossible. Open standards processes can lead to 'design-by-committee' syndrome, but can also greatly increase adoption.

2. Successful systems are those which best manage the tradeoff between hiding implementation and yet still permitting exploration and understanding. The tradeoff becomes clearer as the underlying technologies, such as HTTP, are themselves better understood and standardized.

3. Service orientation, the separation of business logic and application code, and the ability to separate invoker from consumer are important mechanisms for handling application complexity and promoting reusability.

4. Treating remote resources as though they were local is misleading. There are certain properties of local resources, such as latency, that are very difficult to ignore. One of CORBA's problems was its insistence on this principle. Quality of service properties are important to consider. Web services specifications, such as WS-ReliableMessaging, are attempts to address this.

What does the future hold? Based on this historical study some educated guesses suggest themselves. For one, the notion of distributed computing is so useful that it will likely have a growing influence on software engineering. Even today, many users access remote services for everyday computing tasks, such as checking email. With mobile computing playing a larger and larger role, this will likely expand. Evolvability in distributed computing protocols will continue the process of stepping away from lower-level details like transport and security, focusing instead on separation of concerns and sound application design.

In our discussion of the requirements problem, these lessons learned can be seen as reactions to specific requirements problems. For example, vendor lock-in prevented organizations from using the optimal solutions (domain assumptions in the form of constraints restricted certain solutions from being chosen, for example, due to contractual obligations). That is not to say that sub-optimal solutions may not be chosen; while a solution to the requirements problem certainly should be optimal, social processes (”nobody ever got fired for choosing IBM”) may prevent that.

4.2.5 Conclusions From Software Histories

What does the evolution of distributed computing protocols have to say about the practices of software engineering? This section aimed to “deepen insight into the types of activities, methods and tools required, identify and determine those likely to be beneficial, when and how they should be used and how they relate to one another” [Lehman and Ramil, 2003, p.35]. Conducting a longer-term analysis on the evolution of design requirements for distributed computing has provided insight into all of this. For example, my analysis highlighted the issue of scope of encapsulation. Future designers of software systems can take from this history a clearer understanding of how encapsulation can be both a help and a hindrance. This suggests that the choice of solution from one version to another must be made with a good understanding of the full set of implications of the earlier version. The implementation details hidden in today’s web service protocols, for example, are different than those which were hidden in the early RPC implementations. This suggests that part of designing for evolvability might mean altering the scope of encapsulated details as necessary.

One must consider what mechanism to use for reconstructing software evolution patterns. Here I chose to apply fairly traditional historical techniques, looking at the primary sources and reconstructing a history of change by trying to deduce the rationale behind new proposals. This approach pays little direct attention to the context of the change. Other techniques that might be useful include simulation of older proposals (costly for a project of this scope), constructivist approaches such as actor-network theory [Wernick et al., 2006], or interviews with the people involved.

4.3 Lessons

The preceding sections looked at empirical studies of requirements evolution in two ways. In the first approach, I used data mining to extract requirements from repositories. The result of this data mining showed how non-functional requirements changed over the lifespan of the particular project. The goal was to find the unknown factors that explain this change. I determined that this relationship between a particular project and its NFRs over time was non-linear, and most likely responding to external (non-deterministic) events.

The second approach I used was historical. Using the historical artifacts as primary sources, I constructed a narrative about the development of distributed computing technology. In particular, I was able to illustrate how the changes in background assumptions drove and constrained future approaches:

1. The assumption that operational transparency was important evolved to accommodate the realization that remote operations were fundamentally different;
2. A growing understanding of the importance of security in networked applications;
3. The current belief that operation types should be constrained in favour of focus on data and data models (REST) is a result of the many difficulties experienced in interoperability with CORBA and RPC technologies.

There are several conclusions I draw from these studies. One is that it *is* possible to gather requirements from projects which do not use explicit requirements models. Even without that information, I can extract NFRs with relatively good accuracy. For the distributed computing case study, the high-level requirements were often explicitly discussed in design documents. For example, security was an obvious design consideration for nearly all technological approaches.

Another conclusion is that maintenance is *reactive*, rather than pro-active (as postulated in the literature, by Lehman, for example). Furthermore, at least in the empirical studies of open-source projects, these changes occurred as a result of changes in the priority of quality requirements. This is not the same as maintenance that occurs in response to some bug report; we showed that the changes we found were responses to external concurs about quality beyond corrective change requests. Adapting your software system to reflect new information is driven by the information arriving, and not by being flexible. There were numerous examples in the topic mining study which were clearly driven by errors in operation that then inspired changes to the underlying architecture. For example, a bug report might prompt the realization that the software's reporting mechanisms were not adequate. This then drove a number of subsequent fixes that worked on the reliability NFR. In the distributed computing case study, we can see a good example of the design engineering process that Vincenti [1993] talks about. Trial and error is critical in developing improvements; the failure of CORBA, while painful for those deeply invested in the paradigm, nonetheless provided a useful series of lessons for future technologies.

My interpretation of these two conclusions is that what is essential is a means for keeping track of requirements change over time, in order for maintainers to understand the rationale for a given change. We already have excellent ways of doing this for source code, using version control and ticket systems. However, there is no good way of capturing the changes in requirements problems. For example, most tools version individual instances of requirements items, but not the changes in relations between them, nor how that change impacts the overall solution. . It is left to capable designers to analyze prior solutions and seek to improve on them. If such a tool could include some form of reporting on historical developments, this would in turn reveal to managers and developers the places on which to focus. The tool should be able to quickly analyze large projects; although the source code base is large, with the ISO9126 taxonomy there are merely a handful of cross-cutting qualities that are highly relevant to project success.

In the next chapter, I am going to introduce a case study I conducted, subsequent to the work from this chapter. The case study will shine a light on these issues, but from a perspective that includes formal models of requirements problems. This will in turn lay the ground work for understanding how a possible requirements problem tool might work.

Table 4.7: Selected summary statistics, normalized. Examples from *Usability* and *Efficiency* (performance) for selected products using extended signifiers.

Quality	Project	r^2	slope	N (weeks)
Usability	Deskbar	0.08	-0.97	126
	Evolution	0.14	0.27	515
	Nautilus	0.08	0.29	459
	Totem	0.20	0.63	314
Efficiency	Deskbar	0.00	-0.11	34
	Evolution	0.06	-0.02	439
	Nautilus	0.16	-0.10	420
	Totem	0.10	-0.16	158

Table 4.8: Release window technique, normalized

Project	Quality	Release #	r^2	slope
Evolution	Efficiency	2.8	0.22	-0.44
		2.24	0.05	-0.15
Evolution	Usability	2.8	0.01	-1.23
		2.24	0.48	-15.64
Nautilus	Efficiency	2.8	0.01	-0.19
		2.24	0.06	0.28
Nautilus	Usability	2.8	0.03	-2.50
		2.24	0.22	-12.07

Table 4.9: Quality per project. Numbers indicate normalized occurrences per week.

Quality	Project	Occurrences
Efficiency	Evolution	**0.012**
	Nautilus	**0.026**
Usability	Evolution	**0.192**
	Nautilus	**0.285**
Portability	Evolution	**0.010**
	Nautilus	**0.011**

Project	**Measure**	exp1	exp2	exp3
MaxDB 7.500	Named Topics	281	125	328
	Unnamed Topics	219	375	172
	Total Topics	500	500	500
MySQL 3.23	Named Topics	524	273	773
	Unnamed Topics	476	727	227
	Total Topics	1000	1000	1000

Table 4.10: Automatic topic labelling for MaxDB and MySQL

Chapter 5

Case Study: Payment Cards

This chapter describes an extended example to crystallize the concepts we are discussing in this thesis. I used a well-known industry standard, the PCI-DSS, along with documents describing how that standard is realized in an IT environment, and literature reports to conduct a case study into the requirements for securing payment cards. In particular, my case study focuses on how payment card standards have changed over time. These changes can be seen as unanticipated requirements changes from the perspective of individual payment processors and businesses.

5.1 Overview

Payment card networks, and automated teller machines (ATM), are ubiquitous. However, it was not long ago that these machines were in their infancy. After all, consider just a few of the original requirements necessary to implement a functional ATM [Batiz-Lazo, 2009]:

- interact with people with a variety of backgrounds;
- secure customer trust that machine will not make mistakes;
- account for deposits and withdrawals (in an age when there was no Internet);
- resist tampering;
- identify users (when plastic cards themselves were a new idea);
- read magnetic strips (an equally young technology).

Over time, some key requirements have changed:

- ATMs are connected to a bank network.
- ATMs are in a competitive ecosystem (many ATM companies and locations).
- There are many different hardware, software and operating system versions.
- Other functional differences (e.g., foreign currency exchange, mobile phone credits).
- Adversaries employ rapidly changing and sophisticated attacks.

The important consideration is not just *how many* requirements have changed. We must also identify what implications these changes have for the software-based system. For example, security is clearly always a concern; however, attacks on ATMs no longer (or rarely!) consist of attaching a grapple to a truck and towing the machine away. Instead, attacks are more sophisticated, such as fake card readers or hidden keystroke monitors in ATMs, and canny phishing attacks on employees. These changes impact the design of the system. Requirements tools need to understand what changes are relevant, and which requirements can be maintained.

Furthermore, the requirements tend to drift apart as systems age. This might be due to a change in implementation (new ways of meeting a goal) or a change in the meaning of a requirement [Easterbrook, 1995]. At first there are minor differences, largely at the code level. Over time, however, all that remains common are the highest level system goals (allow customer to withdraw money) and constraints on requirements (qualities). Our tools should say something about this distinction, to prepare for risk management or 'future-proofing' our system designs.

5.2 Standards for Securing Payment Card Systems

An example of *unanticipated* requirements change is found in the need to improve security. In the environment in which ATMs operate, it is increasingly important to secure the transactions between the ATM and the bank networks. The Payment Card Industry Security Standards Council[1] 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 standard takes effect whenever a merchant processes or stores cardholder data. The standard applies to the grocery store that swipes your credit card, the processor which validates the information, and the banks which completes the transaction. Although an industry standard, some governments have adopted it as law.

[1]`https://www.pcisecuritystandards.org`

The focus of this case study is the Data Security Standard (henceforth PCI-DSS) [PCI Security Standards Council, 2010], which specifically applies to transactions between merchant and payment processor. There are other standards which focus on securing PIN numbers, physical security, and so on. The PCI-DSS initially started as a set of optional recommendations from VISA (as the Cardholder Information Security Program) in 2005. Given lax compliance, and numerous well-publicized reports of breaches resulting in credit card numbers being stolen, the PCI-DSS v.1.1 contained penalties for card processors who did not meet certain requirements. This version was released in September, 2006, version 1.2 was released in October, 2008, and version 2.0 was released in October, 2010, and is currently in force. The standard defines 244 requirements in total. The consequences for merchants not meeting the standard are financial: in most cases of data breach, where compliance was not verified, the merchant will be heavily fined by the card company. It should be clear that the standard is always reactive. The new possible attacks is invariably an "unknown unknown" for those responsible for standardizing security measures.

The PCI-DSS standard has as its high-level goal to "enhance cardholder data security and facilitate the broad adoption of consistent data security measures globally", which it achieves with the following sub-goals:

- Build and maintain a secure network
- Protect cardholder data
- Maintain a vulnerability management program
- Implement strong access control measures
- Regularly monitor and test networks
- Maintain an information security policy

An interesting addendum permits the use of *compensating controls*, when the technical specification cannot be met, but the risk can still be mitigated. In the modeling language, these will be alternative implementations, since they represent disjunctive refinements of goals.

The goal for organizations falling under the purview of the standard (that is, any organization which handles payment card information, such as a supermarket or e-commerce payment processor) is to demonstrate compliance with the standard. They can do this by passing a compliance audit which is carried out by any number of security audit firms (needless to say, there is a large industry devoted to the PCI-DSS). Industry reports indicate that organizations suffering breaches were 50% less likely to be compliant [Baker et al., 2010].

The PCI-DSS is an overarching, high-level standard, that necessarily represents only a portion of the requirements for a business processing payment card payments. To study the PCI-DSS as applied to a specific domain, I derived requirements for operating a football stadium from another case study [O'Callaghan, 2007], supplementing with online material where necessary. The primary goal of the stadium is to generate revenue, but since sales are (partially) conducted using payment cards, the PCI-DSS standard must be followed.

5.3 The Payment Card Model

PCI-DSS is well-suited for representation as a requirements problem. I map requirements as goals, and constraints as domain assumptions (for example, there is a domain assumption that a football stadium must use legacy IBM devices). Domain assumptions also represent contextual constraints for a particular instantiation of the standard (for example, a specific grocery chain). Tasks represent compliance tests in the PCI-DSS, or committed actions in the domain-specific model. A solution to the requirements problem is a series of tasks which can pass the compliance audit; this set of tasks represents a compliance strategy. I make further domain assumptions about the refinement of goals into (eventual) tasks. Example 1 illustrated this.

The figure in Fig. 5.1 captures a larger instance of this example. The stadium model captured had 44 nodes and 30 relations; the PCI-DSS had 246 goals, 297 tasks/tests, and 261 implications.

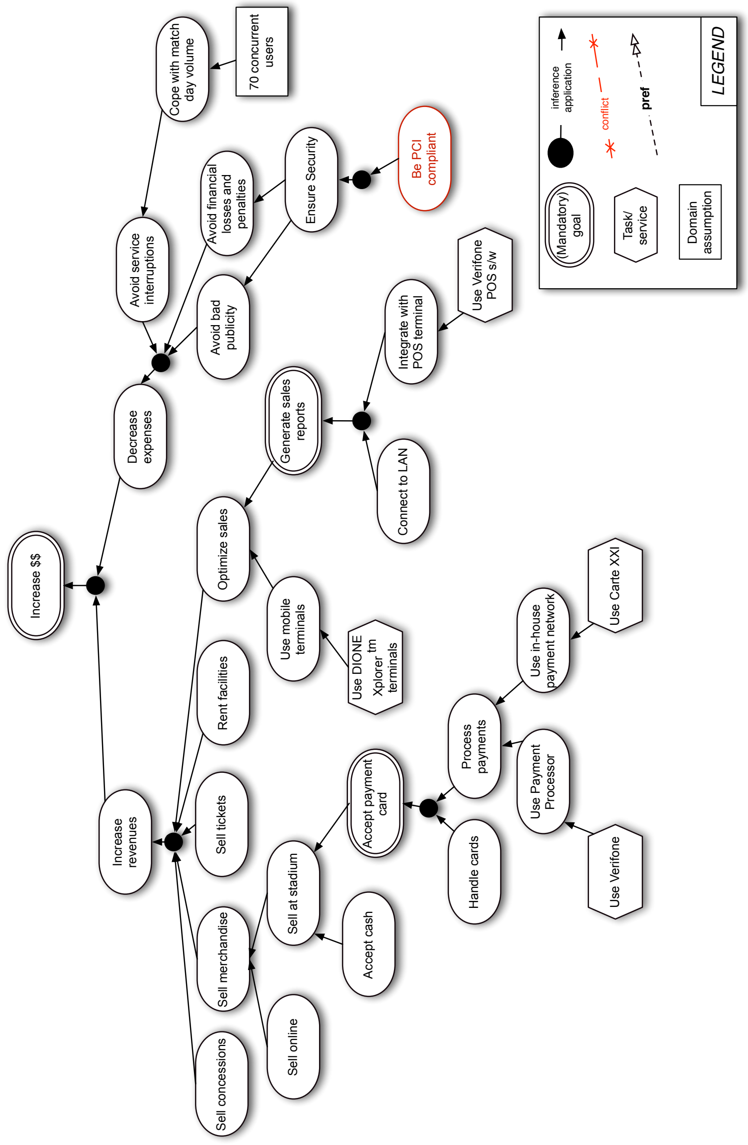


Figure 5.1: The PCI-DSS as applied to a soccer stadium.

5.4 Change Scenarios

The unit of analysis in this case study is a specific change to the PCI-DSS requirements. There are a number of cases that are of interest; I selected the ones that follow based on relevance to the football stadium scenario. For example, one change is the requirement that data be encrypted using Triple-DES, a new encryption standard. Each change is redefining the instance of the requirement problem we must consider when attempting to find solutions. Recall from Chapter 1 that the **Requirements Evolution Problem** is the search for a new solution to a revised requirements problem such that the new solution is in some way related to the original solution.

In the DSS there are multiple types of changes we must accommodate. One type of change is a switch between alternatives, which are variations [Liaskos et al., 2006]. There is little variation in the high-level requirements the PCI-DSS defines, in that a compliant organization must comply with them all. Variation does exist in three other places, however. One is that some requirements, which are preferred requirements in Techne, exist as *best practices*, which are recommended requirements (in that they are likely to become mandatory in the next iteration of the standard).

At the domain assumption level, there are clearly variations *between* organizations. For instance, the context of applying the PCI-DSS to a soccer stadium will be different than for an e-commerce payment processor. This leads to a variation *within* an organization in *proving* compliance (i.e., selecting a compliance strategy). In PCI-DSS these are known as **compensating controls**, and they define solutions for proving compliance where domain assumptions in the standard are invalid. Some examples of a compensating control (numbers in parentheses refer to the PCI-DSS standard, v2):

1. In environments that cannot prevent multiple root logins (requirement 8.1), the organization is permitted to use SUDO (a command to give an ordinary user full but temporary privileges), just as long as the system carefully logs each access. The reason to prevent root login is that the use of a super-user account is opaque, without this control: it is not clear what physical access is behind the root account.

2. If the organization is using older technology, which cannot render cardholder data unreadable (requirement 3.4), then a compensating control is to use a combination of network segmentation, IP access restrictions and injection attack defences.

3. If using secure coding standards (requirement 6.5) for *all* applications is not possible (for instance, a legacy application which does not support a certain best practice), then the standard permits a company to show they use industry standard blackbox testing to verify the application is (more

or less) secure.

4. If a company cannot use Virtual Private Networking (requirement 2.3) to access machines remotely, such as a server host with only FTP access, one can compensate by encrypt key information using e.g. PGP messaging and email.

5. In the example I used in §2.1.2, we saw that it wasn't always possible to use only one server type per machine (requirement 2.2.1). In this case, a compensating control is to show that you have used virtualization, or there are different system accounts per server (e.g., a different user for Apache than for MySQL).

6. If an organization finds it cannot reasonably limit physical access to the data (requirement 9.1), than a compensation is to use rotating key codes to open server room doors and maintain an audit log.

7. For any other circumstance, the PCI-DSS permits the organization seeking compliance certification to ask for a variance.

The interesting aspect of compensating controls, from the perspective of mapping it into a requirements problem formulation, is that they are by definition unique to the context in which they are applied. This suggests that "standard solutions" may not be as standard as hoped. On the contrary, it seems likely, given the large number of domain assumptions which make up the context of each application of the PCI-DSS, that there will be no two compliance strategies that will be identical. This in turn makes it even more important to support incremental changes to the compliance strategy, since the initial strategy will be highly customized to the individual context.

5.4.1 Alternative Compliance Choices

A limited amount of variability exists at the level of choosing compliance goals. For instance, in PCI-DSS requirement 2.2a, the standard suggests the use of one SANS, NIST or CNS for hardening standards. This implies that only one of the three is necessary to prove compliance. Choosing the appropriate standard is part of solving the requirements problem, and will depend on, among other things, the geophysical location and programmer/manager familiarity. Another is the choice between a one-way hash, a one-time pad, PKI or truncation to render cardholder data unreadable (requirement 3.4). Finally, the standard allows leeway in deciding exactly how to perform user authentication (requirement 8.2): either using password or some form of two-factor authentication (such as tokens, smart cards, or biometrics).

5.5 PCI-DSS and Evolution

The previous examples were all variations: they represent alternative means of achieving compliance. Solving the requirements problem will mean identifying the most appropriate compliance strategy for a particular organization. The next aspect of the standard I would like to focus on is the case where the standard itself is changing (in addition to the inevitable change happening within an organization, i.e., changing domain assumptions). What is changing over time in the standard? Answering this question will provide some insight into the importance of managing unanticipated requirements change.

Let us begin by classifying changes into three categories. The standard can expand (add new requirements); it can contract (remove a requirement); or it can be revised (alter an existing requirement(s)). Each change has implications for finding solutions to the requirements problem.

5.5.1 Expansion

Expansion is the addition of a formula to the requirements problem captured as a logical theory. It is assumed that the theory is consistent with the new information.

1. The first scenario is described in requirement 6.2: that ranking system vulnerabilities is a best practice only until July 2012, at which point it is mandatory. In other words, the best-practice goal is expanded with the addition of the mandatory property.

2. Another expansion is to use tokenization (at the merchant level). Tokenization is the use of a hashing function, such as **md5**, to create an alternative representation of the payment card information that will be useless to hackers. This expands the requirement model with a new alternative solution for securing this information.

3. Finally, the standard was expanded with new requirements detailing controls for dealing with malware that evades antivirus software. Antivirus software works by matching malicious code against a known database. If the code is not in the database, then it is not detected. This expansion adds new techniques for managing this situation.

5.5.2 Contraction

Contraction is the removal of a formula from the requirements problem. Again, recall that both contraction and revision require us to remove enough information to keep the requirements problem consistent. In this case, that might mean removing information that had initially justified the contracted information.

1. A typical approach to simplifying corporate IT costs is to offload responsibility for handling payment card data to a third-party. Because the PCI-DSS only applies if your organization directly handles that data, eliminating card data from the merchant zone and using a payment processor to manage transactions alleviates you from complying with the PCI-DSS.
2. The PCI-DSS evolved the requirement to use Network Address Translation to do IP masquerading. This requirement was removed in favour of requirements to either use firewalls or RFC1918 addressing.
3. The change from version 1.2 to version 2.0 of the PCI-DSS removed the ability to use the highly insecure Wireless Encryption Protocol. WEP was the subject of many breaches, so the requirement was dropped. Instead, organizations must use WPA-PK (another encryption standard) as of June 2010 (requirement 4.1.1).

As you can see, many contractions are accompanied by an addition of new requirements. This reflects the dual nature of revision and contraction.

5.5.3 Revision

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 (true of security in general). 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.

5.5.4 Conflicts in the Standard

The final aspect of the PCI-DSS that is relevant to this thesis is the presence of conflicts between portions of the standard. Like any standard, the PCI-DSS contains ambiguities and omissions. The standard also contains some conflicts, which are not necessarily clear in the text, and include possible conflicts with the domain-specific requirements of the organization. Often, the presence of these conflicts is the motivation for designing a compensating control. Some conflicts include:

1. The need to use tokenization and obscure payment card information conflicts with the business need to mine transaction data. Many businesses are used to knowing their customer's home billing address, for example.

2. Centralized identity management is required in the PCI-DSS, but legacy systems cannot have centralized identity management. So-called Federated Identity Management is not available on systems such as Tandem, AS400, MS DOS.

Many more conflicts exist; the compensating controls discussion lists more.

5.6 Conclusions

The PCI-DSS case study has highlighted the importance of changing requirements. A single change in the wording of the standard could result in very costly changes to the compliance strategy for a particular business. We have seen how the standard can expand, contract, and revise its requirements over the course of three versions. This is to say nothing of the changes that are ongoing in the specific organization.

Because the ultimate task in complying with the PCI-DSS is to pass an audit, there is a need to outline for the organization's leadership the necessary strategy to take in passing this audit. Management must consider many questions. For example, should the organization manage payment card information internally, and become responsible for card security, or offload this duty onto a third-party, and cede some control over information. Another question might be whether to upgrade legacy operating systems to newer versions, which can better support modern security practices, but at great cost.

In order to support these types of questions, there is a need for a tool which can a) know about both the PCI-DSS requirements and the organizational context; and b) provide optimal compliance strategies for decision-making. In the following chapter, I will introduce a proposal, called the REKB, which can do both of these things.

Chapter 6

A Functional View of Requirements Problems

So far, this thesis has presented evidence in support of the need to manage requirements problems. I identified a requirements problem as being the search for sets S which are subsets of T, the existing, identified tasks, which would satisfy the requirements, goals G, given the domain assumptions D. S can even be empty if D alone satisfies the goals. The empirical studies showed that requirements exist throughout the software lifecycle. Changes in any aspect of the problem, e.g., in the assumptions D, demand that developers perform adaptive or perfective maintenance in order to continue to satisfy the requirements problem. We can re-state the requirements problem as the search for tasks T and refinements/realizations/constraints R that can be added to the world knowledge/domain assumptions D, such that requirements, captured as goals G, are satisfied, i.e.,

$$D \cup T \cup R \vdash G \tag{6.1}$$

What is not clear, however, is how support the need to understand *what* maintenance to perform. As we saw in §3.2, industry approaches have typically not been sufficient to do this. They cannot provide guidance as to what maintenance tasks ought to be undertaken in order to satisfy the requirements problem once again.

According to the taxonomy of Wieringa et al. [2006] the preceding chapters amounted to *problem investigation*. Therefore, the next step, according to that taxonomy, is to design a solution to the problem. In this chapter, I introduce a functional specification in order to be as flexible as possible, and avoid over-constraining possible implementations for the solution. Subsequent chapters (Chapters 7

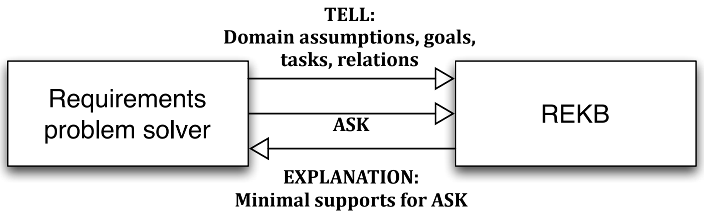

TELL:
Domain assumptions, goals, tasks, relations
Requirements problem solver
ASK
REKB
EXPLANATION:
Minimal supports for ASK

Figure 6.1: The relationship between an REKB and the problem solver

and 8), cover the particular *solution design* decisions I made for this problem, as well as a discussion of *solution validation*, the subsequent stages of the engineering cycle according to [Wieringa et al., 2006].

Recall from §2.1.2 that there are two parts to solving requirements problems. The first step is to identify all *candidate solutions* to the problem, that is, all sets S which satisfy the mandatory goals. The second step is to select from among those candidates a solution which is optimal, according to some definition of optimal and some decision rules.

Two components will handle this: one component, the *problem solver*, will be responsible for understanding decision rules and interacting with the user. The other component, what I have called a *requirements engineering knowledge base* (REKB), is used by the problem solver as a tool for: (i) storing the information gathered during requirements elicitation and domain modeling, as well as justifying problem decomposition; and (ii) asking a variety of questions that can help users compute and compare alternative solutions.

Figure 6.1 shows the relationship between the REKB and the requirements problem solver. I emphasize that what the problem solver does with the basic kinds of answers received, i.e., which solutions it chooses, remains separate. Solving the requirements problem is not the same as operations on the REKB, and answers from the REKB may, or may not, be relevant to solutions. This separation is quite useful if there are different decision criteria for the problem solver. I discuss some of these in §7.2.

To fill out this schema, I must provide details of the capabilities of the REKB. I follow Levesque's seminal account of knowledge bases (KBs) Levesque [1984], which starts with an abstract data type view of KBs that hides implementation details. To do this, what is needed is **a)** a list of *operations* with their syntactic signatures, and **b)** an implementation-independent *specification* of the effect of each operation. Of course, this must be completed by an implementation.

As stated in Levesque [1984], "the capabilities of a KB are strictly a function of the range of questions it can answer and assertions it can accept." Therefore query answering is specified solely based on what has been told. The "knowledge level" approach to specification, advocated by Levesque, to use logical formulas as key languages to interact with REKB, and model theory to specify question answering. In this thesis I am not always able to use a model theory to generate semantic characterizations, but the principle of separating syntactic manipulations of the members of a language from the means to achieve those manipulations remains. The advantage of this approach is that to solve the requirements evolution problem, one can choose reasoning engines according to various independent criteria (e.g., speed, cost, completeness), as long as they achieve the specification.

This chapter looks at the nature of this specification in detail, examining the components of the REKB, the operations it must support, and focusing in particular on the case of evolving requirements. I begin by introducing the core definitions of the REKB (§6.1), and then introduce operations on the REKB for adding or altering data (§6.2). Query operations are more involved: I describe the two major kinds of queries necessary for solving requirements problems in §6.3. To manage the case of changing requirements, I introduce operations to act on these problems, including deriving minimal changes.

This chapter concludes with a look at worst-case complexity for these operations, and some similar work on these ideas (knowledge level definitions were surveyed in §2.3). The majority of this chapter is based on work discussed in [Ernst et al., 2011].

6.1 Definitions

I now introduce the languages used to operate on the REKB. When working with requirements, two abstract families of operations are necessary: ones for adding information of any form listed in the requirements problem definition; and extracting information in the form of solutions to the problem.

First, let us recall the ongoing payment card example I introduced in §2.1.2. A rudimentary graphical representation is captured in Fig. 6.2.

Adding information and extracting information requires (several) TELL and ASK operators, with associated languages $\mathcal{L}$. The general form of the operators will be:

$$\textbf{TELL} : \text{REKB} \times \mathcal{L}_{tell} \rightarrow \text{REKB}$$
$$\textbf{ASK} : \text{REKB} \times \mathcal{L}_{ask} \rightarrow \mathcal{L}_{answer}$$

The basic logical language I use to express requirements ($\mathcal{L}_{tell}$) is quite simple: propositional Horn logic built up from atoms. The language syntax is captured by

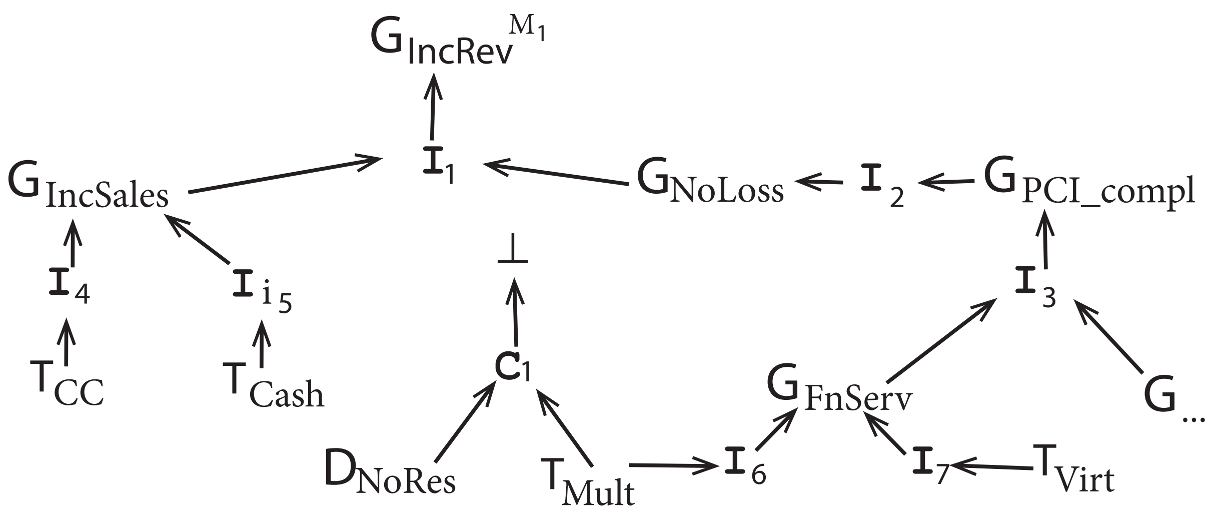


Figure 6.2: A simple requirements problem showing requirements, tasks and justifications.

$$
\begin{aligned}
\textit{formula} ::= {} & \textit{atom}, \ | \\
& (\bigwedge_{i=1}^{n} \textit{atom}_i) \rightarrow \textit{atom}, \ | \\
& (\bigwedge_{i=1}^{n} \textit{atom}_i) \rightarrow \bot
\end{aligned}
$$

Horn clauses are sufficient for my purposes, as they can represent Techne concepts of AND/OR decomposition and refinement, as well as the notions of conflict.

There are three kinds of formulas, as per the definition of the requirements problem, using values from the domain SORT = $\{goal, task, domain\ assumption\}$. GOALS, TASKS and DAS are shorthand for the sets of formulas of the respective sort. Formulas become well-formed (wff) by first assigning each atom a unique value from SORT, and then, if the REKB should respect a given methodological view, specifying how sorted implications can be assembled (e.g., no goals may imply tasks). Sorted implications are members of DAS.

Finally, in order to make it easier to refer to wffs, and represent requirements as graphs, I will attach *labels* λ in front of formulas ψ, to obtain *labelled wffs* $\lambda : \psi$.

A note regarding cycles: I take the bottom-up or fixpoint reasoning approach. That is, at any given reasoning step, we are interested in what new information can be inferred from the available choices. Cycles do not add new information once they have been traversed once. In some sense such cycles are requirements problem 'model smells' (i.e., poor modeling choices) in that they represent situations with no possible resolution.

6.2 TELL Operations

According to the above syntax, one needs to introduce the atoms allowable in the language. For this, I distinguish a symbol table REKB.ST in the REKB, separate from the collection of formulas REKB.TH in it, and provide an operation for declaring new atoms:

Operation 1 — DECLARE-ATOMIC

Domain: REKB × ATOM × SORT

Co-Domain: REKB

Effect: Add the atom to the symbol table REKB.ST, with the appropriate sort.

Throws: Exception if the atom is already declared.[1]

Note that a declaration just introduces a symbol – it is not the same as asserting it to be true. The latter is achieved for any formula by:

Operation 2 — ASSERT-FORMULA

Domain: REKB × WFF × SORT × LABEL

Co-Domain: REKB

Effect: Add the labelled formula to the theory REKB.TH

Throws: Exception if REKB.TH becomes an inconsistent theory, in the sense that REKB.TH $\vdash \bot$.[2]

In practice, we would probably accept wffs as being described in terms of their component labels. This results in the graphs being built directly. For now, I restrict the asserted members of the knowledge base theory to be elements of D and R from the requirements problem (i.e., domain assumptions or refinements/conflicts). Additional formulas that are implications can be added to R and D. Domain assumptions must be included because, at least for now, such atoms are universally true, and thus always relevant. We include refinements and conflicts because they form the set R, mentioned above, and presumably reflect some domain knowledge about how requirements interact.

In some requirements problems we may wish to speculate about certain low-level goals being true or tasks having to be carried out; e.g., "suppose goal g_1 were achieved; what else is necessary to achieve top-level goal g_0?". In that case we may *"hypothetically"* assert these atoms by including them in R ("r: goal g_1 is achieved"), which makes it appear that we *assume* that the corresponding goal/task has been

[1]Exceptions leave the REKB unchanged, and provide a cleaner way to specify special error cases than **return** values. I omit obvious exceptions henceforth.

[2]This could be an exception if we want a particularly simple and efficient subcase where the entire set of actions is a (non-minimal) solution.

achieved.

Note that exceptions only signal special cases and may return useful information. There is no hierarchy of procedure calls as in a programming language. Finally, we need to be able to distinguish formulas (especially, but not exclusively goals) that are mandatory, etc., using

---

Operation 3 — ASSERT-ATTITUDE

Domain: REKB $\times$ WFF $\times$ LABEL $\times$ $\{$*mandatory, preferred*$\}$

Co-Domain: REKB

Effect: Add to the (label of the) formula an indication of its optative nature in the symbol table.

Throws: Raise exception if the label already has an opposite attitude asserted.

Example 3. *Recall from §2.1.2 the example, repeated in Fig. 6.2. In this context, we could introduce* G_{IncRev} *by executing:*

DECLARE-ATOMIC(``The system shall increase revenue.", goal, G_{IncRev});

ASSERT-ATTITUDE(G_{IncRev}, *mandatory*);

Finally, as part of evolution, clearly one needs inverse operators for DECLARE-ATOMIC and ASSERT-ATTITUDE, to be called RETRACT-DECLARE and RETRACT-ATTITUDE, in order to revise the symbol table appropriately. In the case of assertions, we only allow retracting formulas that have been previously explicitly asserted (and hence labelled), using operation RETRACT-FORMULA(LABEL).

6.3 ASK Operations

ASK operations retrieve information about the requirements problems stored in the REKB.

In any practical situation one would want to retrieve information stored about atoms and formulas in the symbol table REKB.ST. The logical specification (and subsequent implementation) of these operations is obvious. What distinguishes these is that they do not rely on deduction from REKB.TH, only on REKB.ST. Some of these operations include:

- the ability to retrieve the labels associated with a given SORT,
- the formulas in which a given atom participates,
- the attitude on a given atom (i.e., mandatory, preferred, or undefined).

6.3.1 Goal Achievement

One question users of the problem solver will have is to know whether a goal can be achieved using some set of tasks based on the domain knowledge/refinements accumulated so far in the REKB. Given a set V, we use $\wp(\mathrm{V})$ to represent the powerset (set of subsets) of V.

Operation 4 — ARE-GOALS-ACHIEVED-FROM

Domain: REKB $\times$ assumeT $:\wp(\text{GOALS} \cup \text{TASKS}) \times$ concludeG $:\wp(\text{GOALS})$

Co-Domain: Boolean

Effect: returns true iff REKB.TH $\cup$ assumeT $\vdash \bigwedge$ concludeG

Throws: throws exceptions if assumeT or concludeG are inconsistent with background knowledge, i.e. REKB.TH $\cup$ assumeT $\vdash \perp$ or REKB.TH $\cup$ concludeG $\vdash \perp$.

If the operation will return False due to inconsistent goals, it should inform the user that this is not due to the current state of the REKB, but rather the inputs to the operation. It could have variants such as defaulting to the set of all tasks if assumeT is omitted, or always including mandatory goals in concludeG.

This operation supports a fine-grained exploration of what can be achieved using specific actions or from certain subgoals. By itself, it does not solve the requirements problem directly, (though it could be used in a brute force enumeration, to do so); on the other hand it has a much lower computational complexity for our Horn-logic language. In other goal-oriented reasoning approaches, this operation is known as label propagation [Giorgini et al., 2003]. Note that with the parameter assumeT we are implicitly asserting certain tasks and/or subgoals to be True, and then attempting to show that the members of concludeG hold.

Example 4. *Using example 3 from before, a sample call to this operator might be ARE-GOALS-ACHIEVED-FROM*$(\{T_{CC}, T_{Virt}\}, G_{IncRev})$*, to which the answer is TRUE.* ■

Example 5. *What might cause ARE-GOALS-ACHIEVED-FROM to throw an exception? Suppose that in our REKB we have asserted the following sentences:* $\{A \rightarrow C, C \rightarrow G, C \wedge D \rightarrow \perp\}$*. If we now call ARE-GOALS-ACHIEVED-FROM*$(\{A, D\}, G)$*, then we will cause an exception, since asserting* A *and* D *allows* $\perp$ *to be derived.*

6.3.2 Finding Candidate Solutions

The next operation is to support what has been called "backward reasoning" in the Requirements Engineering literature (e.g., [Sebastiani et al., 2004]). Backward reasoning sets some high-level goals as desiderata, and determines which tasks can accomplish those goals.

Operation 5 — MINIMAL-GOAL-ACHIEVEMENT

Domain: REKB $\times$ concludeG $:\wp(\text{GOALS})$

Co-Domain: $\wp(\wp(\text{TASKS}))$

Effect: returns a set that contains all sets S of tasks (or goals in REKB) such that REKB.TH $\cup\, S \models \bigwedge$ concludeG, no subset of S has this property, and REKB.TH $\cup\, S$ is consistent.

Throws: throws exception if concludeG is inconsistent with REKB.TH.

These answers are essentially *abductive explanations* of how the goals can be achieved from the tasks [Levesque, 1989]. In the case when concludeG contains all mandatory goals, this provides "minimal candidate solutions" to the requirements problem according to the terminology in [Jureta et al., 2010]. The

Example 6. *A call to this operator might be MINIMAL-GOAL-ACHIEVEMENT($\{G_{IncRev}\}$), to which the answer is $\{\{T_{Virt}, T_{Cash}\}, \{T_{Virt}, T_{CC}\}\}$. Which set to choose to implement is not determined by the operator. For example, we may choose the set with fewer members, or we may use a cost function to determine the less costly set.* ■

Finally, in the requirements problem we are interested in optimality with respect to the people communicating the requirements for the new system (stakeholders). In that context, the stakeholder may not be content with a subset-minimal implementation that satisfies the mandatory requirements. Rather, he or she is interested in implementations which also satisfy other, non-mandatory goals. Furthermore, while still subset-minimal with respect to tasks, I add the constraint that the set of goals is maximized. This answers the question, "If I wish to accomplish the following extra goals, in addition to my mandatory requirements, what are the minimal sets of tasks I must perform?" Each solution is still required to be consistent, of course. This requirement can be satisfied by using the following operation:

Operation 6 — GET-CANDIDATE-SOLUTIONS

Domain: REKB $\times$ mandG : $\wp$(GOALS) $\times$ wishG :$\wp$(GOALS)

Co-Domain: maxC:$\wp(\langle \mathsf{solnT}, \mathsf{satG}\rangle)$ where solnT :$\wp$(TASKS) and satG :$\wp$(GOALS)

Effect: maxC is a set of tuples such that 1) REKB.TH $\cup$ solnT $\models$ satG with solnT $\cup$ REKB.TH consistent, 2) satG has all goals in mandG (which contains those goals for which the REKB has been TOLD ASSERT-ATTITUDE(*mandatory*)), and as many goals in wishG as possible, and 3) solnT is subset minimal to achieve the above.[3]

Throws: throws exception if wishG is inconsistent with REKB.TH.

Example 7. *Using the previous example, let us assume that the user has created as wishG all the remaining goals in the example. The user now calls GET-CANDIDATE-SOLUTIONS($\{G_{IncRev}\}$,wishG). We know the minimal solutions for the mandatory goal G_{IncRev} are the sets $\{\{T_{Virt}, T_{Cash}\}, \{T_{Virt}, T_{CC}\}\}$. Now the question is how to grow the set of preferred goals either solution satisfies, while remaining consistent and minimal. We start with the first minimal solution. We can attempt to add T_{Mult}, but this is subsumed by the existing solution, and in any case, conflicts with a domain assumption. For the second minimal solution, our only other choice was the alternative T_{Cash}, which again is subsumed, and hence not minimal, while satisfying the same number of goals. If, however, T_{CC} satisfied another goal in addition to $G_{IncSales}$, for example it satisfied a security goal (since less cash must be stored), then adding T_{CC} to a solution would be useful for the stakeholder.* ■

6.3.3 Deciding Among Candidates

The previous operations find a set of candidate solutions. The next step in solving the requirements problem is to decide between these candidates. There are many possible decision criteria. One possible criterion is the number of preferred goals a solution includes. One would want to determine which sets of tasks identified as solutions maximize the number of preferred requirements they satisfy. Another criterion is the number of preference relations a solution satisfies. In §7.2.1 I discuss the specific implementation of this in more detail.

[3]The set wishG may default to those goals which have been ASSERT-ATTITUDE(preferred) for simplicity.

6.4 Operations for Requirements Evolution Problems

The previous section considered the case of a static REKB. In this section, I look at what operations on the REKB are necessary to support the Requirements Evolution Problem.

6.4.1 Finding Minimal Change Tasks

The following generalized operator is intended to help find solutions to a subset of problems in incremental evolution.

Operation 7 — GET-MIN-CHANGE-TASK-ENTAILING

Domain: REKB $\times$ goalsG $:\wp(\text{GOALS}) \times$ originalSoln $:\wp(\text{TASKS}) \times \mathsf{DIST} - \mathsf{FN}$

Co-Domain: $\wp(\wp(\text{TASKS}))$

Effect: Return the set of task sets S which solve REKB.TH $\cup$ $S \models$ goalsG, and which minimize DIST-FN$(S, originalSoln)$.

Example 8. *Assume our stadium implemented $\{T_{Virt}, T_{CC}\}$ from Example 6. The PCI-DSS consultant has recommended the use of a tokenization framework in order to reduce the chance of someone hacking into a mobile terminal and obtaining payment card information. To do so we must add the implementation of tokenization (T_{md5}) and update the software on the payment processor servers to manage tokens instead of credit card data.*

We update our REKB *with this new information: declare atoms $G_{PrData}, G_{Token}, T_{md5}, T_{UpServer}$; assert wffs representing the refinement of G_{PrData} by the goal of using tokenization, and G_{Token} by $T_{md5}, T_{UpServer}$. Figure 6.3 reflects the new* REKB. *A call for this operator might be* *GET-MIN-CHANGE-TASK-ENTAILING*$(\{\{G_{IncRev}\}\}, \{\{T_{CC}, T_{Virt}\}\}, \Pi_1)$*, to which the answer is* $\{T_{md5}, T_{UpServer}\}$*. In this case* Π_1 *is returning the minimal change solution, since this involves implementing just two new tasks. Other distance functions might return different results, of course.* ■

A Note on Simplifications – In the REKB, I make the simplifying assumption that no tasks or goals can be individually asserted. Instead, these could be established by adding to the background theory R (that is, the set of refinements) some assumptions which assert that a particular goal is satisfied. In particular, we would want some well-specified mechanism which can somehow mark the implemented tasks (e.g., by asserting them in the REKB.TH). This will allow for finer-grained constraints on the solution returned. Currently, this is handled outside of the REKB, but ideally would be part of the operator definition. This is particularly important when we discuss inconsistent REKB in Chapter 9.

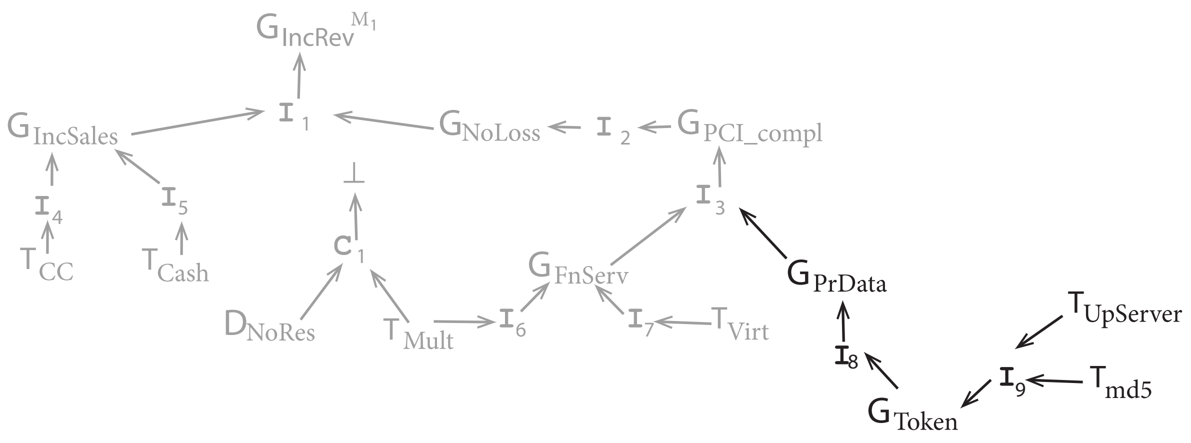


Figure 6.3: The revised REKB from Figure 2.1 showing added elements.

Since we adopt a credulous paraconsistent semantics, we must ensure that the particular maximally consistent set we choose not only satisfies the requirements problem, but does so by considering the existing set of (asserted) tasks and/or goals.

6.5 Complexity

I briefly mention the most straightforward implementations (as upper bounds) and known complexity results (as lower bounds) for implementing these operations.

The TELL and UNTELL operations are just book-keeping ones, while ASSERT-FORMULA checks whether the addition of its argument results in REKB.TH being inconsistent. Although in general this is an NP-complete problem, it is well-known [Dowling and Gallier, 1984] that for Horn clauses this can be done in time linear in the size of REKB.TH.

The ARE-GOALS-ACHIEVED-FROM operation could again be implemented as propositional theorem proving, but using a proof by refutation, the ability to achieve a single goal can be found by testing the consistency of a Horn theory, which once again can be performed in linear time.

The MINIMAL-GOAL-ACHIEVEMENT operator is no longer a propositional entailment question, since we are looking for *minimal sets of tasks* that entail a goal. This is the *abduction* problem. It is is much more difficult because there may be exponentially many solutions, and even if there is only one solution, it is known to be NP-complete to find it even for Horn clauses (see Eiter and Gottlob [1995] for a good summary of propositional abduction complexity). While our experiments (below, in §7.4.2) show that this worst-case limit is rare, it is possible to construct pathological examples which render the reasoning intractable. More empirical experience will provide further insight into the viability of the approach.

Note that our operator is finding minimal *subsets*, and not minimum cardinality sets (a slightly harder problem). Although the discovery of all minimal subsets means finding the minimum cardinality sets is simple, finding *one* minimal subset may be cheaper than finding the minimum one, because there may be exponentially many minimal ones to compare. On the other hand, merely focusing on task set size is not necessarily relevant; we want measures of "optimal" to apply to many things.

The GET-CANDIDATE-SOLUTIONS operation requires several calls to ARE-GOALS-ACHIEVED-FROM, which is linear, but also a single call to MINIMAL-GOAL-ACHIEVEMENT. Furthermore, the operation is potentially exponential in the size of the set wishG, in the event that we wish to reduce the set to only those members which can be satisfied.

The GET-MIN-CHANGE-TASK-ENTAILING operation can be implemented by finding all ordinary solutions (using MINIMAL-GOAL-ACHIEVEMENT), and then filtering them through the condition DIST-FN(_,X_0). Its cost is therefore the same order of magnitude as MINIMAL-GOAL-ACHIEVEMENTsince the tests are set-theoretic operations which take polynomial time.

6.6 Related Work in Requirements Engineering KBs

Within the scope of RE, the idea of an REKB is most accurately categorized as "requirements management".

Many of the operations I identified in §8.4 were introduced in Cleland-Huang et al. [2003]. That paper takes a traceability approach to managing requirements. In that work, seven operations are defined (for which traceability events occur). These operations are: **create** a new requirement; **inactivate** a requirement; **modify** an attribute of a requirement; **merge** two or more requirements; **refine** a requirement with an additional part; **decompose** a requirement into separate parts; **replace** one requirement with another. However, the focus was on using these events to spawn traceability actions, and not to solve requirements problems.

The Requirements Engineering System of Lormans, outlined in Lormans [2007], also argues for a requirements management framework. Case studies in the embedded systems domain, e.g., Lormans et al. [2004], motivate the need to manage changing requirements throughout the software lifecycle. The principal difference is the focus on traceability links generated from latent semantic indexing, where we instead focus on the model versioning aspect.

6.6.1 KAOS

The KAOS goal-oriented requirements engineering framework, first described in Dardenne et al. [1993], has been the focus of several papers that arguably consider a specification-level approach to the requirements problem. KAOS consists of an acquisition methodology, a semantic network representation of the requirements problem, and a formal representation based on the linear temporal logic [van Lamsweerde and Willemet, 1998]. KAOS introduced a very expressive conceptual model for requirements analysis, including agents, goals, requirements, and events. The focus of KAOS was goal satisfaction, and the production of a detailed, temporally ordered specification that would satisfy the requirements. Solving the requirements problem in KAOS is to complete the KAOS acquisition methodology, beginning with identification of system goals, and resulting in (possibly many) assignments of tasks to agents. Variants of KAOS have been proposed to handle partial satisfaction using numeric weights [Letier and van Lamsweerde, 2004]; to formally describe the nature of conflicts in RE [van Lamsweerde et al., 1998]; to model obstacles to requirements satisfaction [van Lamsweerde and Letier, 2000], and to reason about alternative solutions for requirements problems [van Lamsweerde, 2009].

The work in this thesis focuses very little on the problem-definition phase, instead assuming these issues are either resolved interactively, by running the problem solver, or by assuming there is some agreement already. My work also does not have the scope to cover temporal operators for specifying temporal reasoning problems (although such might be accomplished using domain assumptions). The more limited ontology of the REKB is reflective of its relative immaturity (KAOS having been created some 20 years ago). The limitations are also reflective of deliberate design choices: that a simple approach is both more tractable for automated reasoning (it is not clear if KAOS's LTL representation is decidable), as well as more flexible for early design work. KAOS and Techne are addressing different portions of the broader requirements engineering lifecycle.

6.6.2 Other RE Approaches

Giorgini et al. [2003] and Sebastiani et al. [2004] introduced the notion of "solving" goal-oriented requirements models, in what I have called Qualitative Goal Modelling (QGM). In the first paper, the focus was on formalizing qualitative reasoning using the notations introduced in [Chung et al., 1992]. There, the ASK question was "can this goal be satisfied given this initial state?", which is the analogue of our ARE-GOALS-ACHIEVED-FROM. In the second paper, the earlier work was extended to look for the minimal sets of *input* goals that must be satisfied in order to satisfy the *desiderata*, or output goals. This is similar to my MINIMAL-GOAL-ACHIEVEMENT operator.

Menzies et al. [1999] proposed some generic questions for evaluating a requirements model. Specifically, they introduced the following five questions as important for reasoning with requirements:

1. Which models provide best coverage of a set of goals?
2. Which models provide fairest coverage of different stakeholders' goals?
3. Which models provide best coverage of sets of conflicting goals?
4. Which models obtain the highest score in covering a set of weighted goals?
5. Which models obtain the highest score for covering a set of goals when weights are added to model properties (e.g. size, complexity, readability, etc)

This work primarily addresses question (1); I do not consider multiple models, nor do I provide for numeric weights. Like this thesis, [Menzies et al., 1999] used an abductive approach to answer question (1).

Finally, our work in Jureta et al. [2010] is a precursor to this thesis. That paper introduced the notion of RE as the process of solving requirements problems. In that work, we first identified the notion of **r-net**, which is similar to the REKB here. From this r-net we identified candidate subnets, that is, conflict-free parts that satisfied the mandatory goals. We formally defined the following questions:

1. What are the conflict-free candidate subnets of the r-net? These are sets of elements in the r-net which do not instantiate the conflict relation.
2. Of those conflict-free subnets which ones are candidate solutions? These are subnets which satisfy the mandatory parts of the r-net.
3. Of the candidate solutions which do best with respect to preferences and preferred elements (which were called options)? This becomes the decision procedure for choosing between candidate r-nets.

This chapter has expanded on these questions to provide a generic interface for solving requirements problems. By doing so, I have largely remained above the details of implementing the operations themselves. In the subsequent chapters, I will focus in more detail on these implementations, and begin evaluating the approach on realistic models. I will show that the typical-case complexity of some of the operations (particularly abductive reasoning), in RE, are tractable for problems we are likely to encounter. Inconsistency is considered in more detail in Chapter 9.

Chapter 7

Solving Static Requirements Problems

The preceding chapter provided a functional view of requirements problems. It outlined abstract operations which will find and select solutions. The next step in a functional approach is to provide implementations of the functional specification, and so this chapter considers ways to implement the aforementioned REKB operations. For the moment I will continue to restrict the discussion to consistent formulations of the requirements problem.

This thesis is concerned with the problem of managing unanticipated requirements evolution. Before we consider how to deal with that problem, however, I must begin with approaches to *finding* solutions to the requirements problem when the problem is static, that is, when our REKB has not been subjected to a change. Understanding this problem is key to ultimately solving evolving requirements problems, which I discuss in the following chapter, Chapter 8.

I will begin by discussing approaches to finding solutions. I will introduce the mechanism used to implement the REKB, leveraging the ideas from reason maintenance systems[1]. I then discuss the problem solver that sits between user and REKB (recall Fig. 6.1). I consider in detail the implementation and algorithms for ARE-GOALS-ACHIEVED-FROM and MINIMAL-GOAL-ACHIEVEMENT.

The next step is to decide on a particular solution, so §7.2 will consider algorithms for doing just this, based on work in [Ernst et al., 2010]. The problem solver not only interacts with the REKB, but also the user, so I will consider how to allow users to specify requirements problems in §7.3. I look at using

[1]While commonly known as truth-maintenance systems, this is not 'truly' reflective of what such systems do [Doyle, 1983].

a domain-specific language as well as a graphical editor.

Finally, it is important to demonstrate the applicability of these implementations, so in §7.4.2 I will turn to empirical validations of these approaches, in particular my approach to the MINIMAL-GOAL-ACHIEVEMENT operator. Here I will show how the payment card case study (Chapter 5) can be evaluated using the implementations in this chapter.

7.1 Finding Solutions

7.1.1 The Assumption-Based Truth Maintenance System ATMS)

The representation of REKB as sets of nodes with Horn clauses linking them to form a graph, i.e., the structure presented in §6.1, is exactly the structure underlying a *truth maintenance system (TMS)* [Doyle, 1979; de Kleer, 1986]. The A(ssumption-based)TMS of de Kleer [1986] is particularly well-suited for implementing the operations introduced in the preceding chapter. It is not, however, the only such implementation choice, and I discuss alternatives in §7.5.

An *explanation* is a minimal set of tasks that justify (entail) a given node. The advantage of ATMS is that it

1. provides (all) explanations for why a particular node is justified;
2. is designed to find *minimal explanations* for each node;
3. the ATMS automatically eliminates explanations containing inconsistencies; and
4. it can incrementally compute new explanations for newly added atoms or wffs. This incrementalism is leveraged in Chapter 8.

The notion of a reason-maintenance system evolved from the need to better support problem-solving in Artificial Intelligence. In particular, when searching for a solution to a problem (for example, N-QUEENS[2]), traditional AI problem-solving techniques had five main shortcomings solved by reason-maintenance [Forbus and de Kleer, 1993]. Particularly relevant for solving requirements problems was the ability to identify justifications or rationale for a particular decision (hence "reason-maintenance"). We often want to know why a certain solution has been rejected or accepted in addition to the decision itself. In addition, reason-maintenance systems could guide back-tracking intelligently, so if a problematic state was reached, the problem-solver would not have to re-search that part of the space.

[2]http://en.wikipedia.org/wiki/Eight_queens_puzzle

The ATMS itself is intended as a storage system for the results of problem solving. That is, an inference engine makes use of the ATMS to keep track of the status of the problem-solving. I use it in much the same manner for solving requirements problems: the ATMS keeps track of what it has been told, and the problem-solver interfaces with the user and draws conclusions from the answers the ATMS provides. We can think of the interaction between the two as a means of recording the process of eliciting requirements (using the CORE ontology of Jureta et al. [2008].) As stakeholders communicate speech acts to the requirements engineer, the problem-solver adds that information to the ATMS, incrementally constructing a representation of the requirements problem. The ATMS captures (exactly) the Techne notion of justification: that is, the directive speech act "The system shall increase sales" (called a *datum* in [Doyle, 1979]) becomes the ATMS *node* $G_{IncSales}$:"increase sales". The ATMS is blind to the content of the node:

> "... [because of separation between problem-solver and truth-maintenance system TMS] there are two inference procedures in the reasoning system both operating on the same expressions but treating them entirely differently. ... The TMS determines belief given the justifications encountered thus far and not with respect to the logic of the problem solver. [de Kleer, 1986, p. 142]"

Reason-maintenance systems distinguish three node properties:

- A **premise** is a node that is universally true (in the universe of the TMS). These might be Techne domain assumptions, although for simplicity, I assume all Techne relations (conflicts, implications) are premises.
- A **contradiction** is a node that is never true. In Techne this is equivalent to the datum $\perp$.
- An **assumption** is a node that is assumed to be true. In Techne this is equivalent to the datum with SORT TASKS.

TMS nodes are connected using **justifications**, which map exactly to Techne implications (i.e. wffs of the form $(\bigwedge_{i=1}^{n} atom_i) \rightarrow atom$). A node in a TMS has no value if it is neither an assumption nor justified by another node; since it is not believed, we can consider it false.

de Kleer [1986] proposed the ATMS, an extension of Doyle [1979]'s (J)TMS. In an ATMS each node has associated a set of possible *explanations*, in which that node is `:IN` (interpreted as true). Explanations are sets of assumptions which ultimately justify that node (i.e., from which that node can be derived from assumptions via definite Horn-rules called justifications.) The *label* for a given node N will take one of three values: if there is no justification for N, the label is said to be empty: $\langle N : \{\}\rangle$; if the node is always :IN, i.e., it is an assumption (in our case, a task), then the label has an empty explanation: $\langle N : \{\{\}\}\rangle$. Finally, in all other cases, the node is labelled with explanations: all sets of assumptions for which it can be derived :IN. For example, $\langle N : \{\{A, B\}, \{C, D, E\}\}\rangle$.

Most importantly, these sets are minimal – no nodes can be removed from such an explanation without losing the full justifications, and the sets are consistent in the sense that no contradictions ($\perp$) can be derived from them. I encode atoms in the REKB as ATMS nodes; atoms with sort TASK become assumptions, and implications or contradictions become justifications and contradictions, respectively, with a special CONTRADICTION node added as necessary. The CONTRADICTION node represent the consequent in Techne's conflict relation $(\bigwedge_{i=1}^{n} atom_i) \rightarrow \perp)$.

Cycles are supported in the ATMS because of short-circuit evaluation. If a node has a label which is the same as a possible new label, evaluation terminates. This is similar to the idea of marking an edge in a graph as visited when doing graph traversals [Dowling and Gallier, 1984].

The ATMS provides low-level operations, acting as a theorem prover. I leveraged existing code for the implementation from [Forbus and de Kleer, 1993]. However, alone it cannot implement all operations described in Chapter 6. That implementation must be extended with a problem-solver module which implements the operations which are not supported in the ATMS itself, such as retraction and handling of preferred nodes. After discussing the ATMS handling of ARE-GOALS-ACHIEVED-FROM and MINIMAL-GOAL-ACHIEVEMENT, I then turn to discussion of how this is supported.

7.1.2 ARE-GOALS-ACHIEVED-FROM

From the specification in §6.3.1, this operation checks if the concludeG set of outputs (or desiderata) are achieved (true) for the assumeT set of inputs (which are tasks or goals). With the ATMS back-end, the simple version of this operation is the case where concludeG is a single goal, and assumeT are only tasks. Then the desired goal will be an ATMS node with an explanation label, and we merely have to verify that this label has at least one explanation that is contained in the set assumeT. It is possible to convert a goal node in Techne into an ATMS assumption using a meta-node which is enabled when the goal is to be assumed true. This mimics the abstraction mechanism of assuming that the goal is achieved *somehow*, without requiring us to provide unnecessary detail. The mechanics of this are similar to the retraction mechanism discussed below.

In the requirements problem, however, it is more common that we will have a set of several goals we desire to achieve, namely, those which are marked as mandatory. In this case, each of the members of concludeG will have an associated set of explanations. Our task is then to find the union of explanations common to all members of concludeG. Alg. 3 shows our approach, with Alg. 1 and 2 secondary functions which determine subsumption relationships and remove subsumed or inconsistent explanations, respectively. The idea is to obtain the same result as if we had introduced a new singleton node representing the consequent of an implication with the members of concludeG as the antecedents.

Those explanations are the same as those for the set concludeG. Comparing these explanations to the set assumeT will then help us to decide the question posed by ARE-GOALS-ACHIEVED-FROM: if any explanation is equal or subsumed by assumeT, then the answer to ARE-GOALS-ACHIEVED-FROM is True.

Algorithm 1: COMPARE-EXP(exp_1, exp_2) - explanations for a node

```
Output: symbol
case $exp_1 \subset exp_2$
    return :21
case $exp_1 \supset exp_2$
    return :12
case $exp_1 = exp_2$
    return :EQ
otherwise
    return nil
```

Algorithm 2: FILTER(E) - set of explanations

```
foreach $e \in E$ do
    // remove subsumed and NOGOOD environments
    foreach $n \in NOGOOD$ do
        // NOGOOD is a table of minimally inconsistent explanations
        result = COMPARE-EXP($e, n$)
        if $result$ = :21 OR $result$ = :EQ then
            $E \setminus \{e\}$
    // remove subsumed and duplicate with COMPARE-EXP
    foreach $e' \in E$ do
        result = COMPARE-EXP($e, e'$)
        if $result$ = :21 OR $result$ = :EQ then
            $E \setminus \{e'\}$
return $E$
```

7.1.3 MINIMAL-GOAL-ACHIEVEMENT

Given a call MINIMAL-GOAL-ACHIEVEMENT(concludeG), if the ATMS created a super-node representing the conjunction of the goals in concludeG, this would allow us to simply read off the answers required. Short of that, we have to perform operations outside the ATMS which combine the explanations of the desired goals, and test them for consistency. Alg. 4 outlines the approach. Again, we compare explanations between the goals in concludeG, this time seeking the smallest (by subsumption) which achieves the

Algorithm 3: ARE-GOALS-ACHIEVED-FROM

```
Input: assumeT :$\wp$(GOALS $\cup$ TASKS), concludeG :$\wp$(GOALS)
Output: Boolean
$I', I := \{\{\}\}$
foreach $n \in$ concludeG do
    // LABEL(x) returns the explanations for node $x$
    foreach $e \in$ LABEL($n$) do
        foreach $i \in I$ do
            $I' := I' \cup (e \cup i))$
    $I :=$ FILTER($I'$)
// now, assess whether some explanation in $I$ either a subset or is equal to
   assumeT
foreach $exp \in I$ do
    test := COMPARE-EXP($exp$, assumeT)
    if $test$ == :S21 OR $test$ == :EQ then
        entailed? := True
return entailed?
```

goals. We are seeking a set of consistent sets of tasks or 'assumed' goals which are either in each label of all members of concludeG, or subsume an explanation in those labels.

7.1.4 GET-CANDIDATE-SOLUTIONS

The final operation was GET-CANDIDATE-SOLUTIONS, whose purpose is to expand on the MINIMAL-GOAL-ACHIEVEMENT approach to finding solutions by attempting to expand the solution sets such that they satisfy additional, preferred goals. Given mandG $= \bigwedge G$, where G again represents the mandatory goals, calling MINIMAL-GOAL-ACHIEVEMENT on mandG will derive the solution sets S_j which are minimal explanations for mandG. There are now two options, representing bottom-up and top-down approaches. If the algorithm begins bottom-up, then it first looks at a $g \in$ wishG, and its explanations, $E_1, E_2 \ldots E_i$. It then considers the union of each explanation for mandG and each explanation for g, i.e., $S_1 \cup E_1, S_1 \cup E_2, ...S_j \cup E_i$. It filters each such set through the minimality and inconsistency tests (i.e. FILTER from §7.1.2), and obtains a new set of solutions which satisfy at least one new preferred goal. It then begins once more with the resulting solutions, adding a remaining member of mandG g, and so on. Algorithm listing 5 below shows the high-level idea, with the call to TEST-SAT implemented as just described.

The top-down approach begins by testing the union of the sets S_j, that is, the minimal explanations for mandG, with the single set of goals in wishG. Ideally, there are no conflicts, and we are finished. Realistically, this is not the case, and we must proceed by removing a single member of wishG and testing the remaining set unioned with each of the S_j. The advantage of this approach is that if it indeed finds a consistent, minimal explanation for $S_j \cup$ (wishG $x), x \subset$ wishG, it may obtain a maximal consistent

Algorithm 4: MINIMAL-GOAL-ACHIEVEMENT

```
Input: concludeG :℘(GOALS)
Output: ℘(℘(TASKS))
entail-sets := {}
foreach pivot in POP(concludeG) do
    meet-exp := exp
    add-exp? := T
    foreach other in goals do
        all-false? := T
        foreach other-exp in LABEL(other) do
            outcome := COMPARE-EXP(meet-exp, other-exp)
            if outcome = :EQ OR outcome = :S21 then
                all-false? := nil
            end
            else if outcome = :S12 then
                meet-exp := other-exp // new smallest explanation
                all-false? := nil
            end
            else if outcome = nil then
                // do nothing
            end
        end
        if all-false? then
            add-exp? := nil // all explanations in this goal were not a match, so the
                pivot exp is invalid.
        end
    end
    if add-exp? then
        APPEND(entail-sets,meet-exp)
    end
end
return entail-sets
```

explanation quicker than the bottom up approach. In any event, both algorithms are exponential in worst-case running time.

A heuristic approach, which I implemented with a slightly different language pQGM, is described in §7.4.1. This uses a local search technique to explore the space of explanations for mandG ∪ wishG and is of course much quicker, at the expense of optimality.

These three operations describe techniques for identifying solutions to the requirements problem. Once these candidate solutions have been produced, the remaining step is to decide between them. This is the subject of §7.2.

Algorithm 5: GET-CANDIDATE-SOLUTIONS, bottom-up

```
Input: mandG :$\wp$(GOALS), wishG :$\wp$(GOALS)
Output: maxC:$\wp(\langle$solnT, satG$\rangle)$
$S$ = MINIMAL-GOAL-ACHIEVEMENT(mandG)
collected = $\emptyset$
foreach $S_i \in S$ do
    collected = collected + TEST-SAT($\{\langle S_i$, mandG$\rangle\}$ for $S_i \in S$)
    // collected is the set of tuples of tasks and goals
end
return collected
```

7.1.5 The Problem Solver

The above operations are easily done within the ATMS. The ATMS cannot support all operations, however, and these are necessarily implemented in the problem solver interface. For one, a problem with an ATMS as an REKB is that they do not normally support retraction. Therefore, when a leaf node (task) t is retracted, i.e., is no longer :IN, the program using the ATMS must filter out explanations that contain t. I simulate this by storing a Techne rule $\lambda : a \wedge b \rightarrow c$ as $\lambda \wedge a \wedge b \rightarrow c$, i.e., by including its label as a conjunct. Then when answering questions based on the solution provided by the ATMS, the possible explanations are filtered to remove all those that contain the label of retracted rules (since they can no longer be derived). Fig. 7.1 shows how this works. When the formula is first added (step 1 in the figure), a 'dummy' node λ is attached to the antecedents. This makes (an) explanation for $C = \{A, B, \lambda\}$. When the formula is retracted (step 2), λ is made inconsistent, and this explanation is removed from the label for C. A similar process works with tasks. To re-enable that formula, (currently) the problem solver re-creates the original justification with a new node λ'.

This causes performance problems since the ATMS calculations grows exponentially with the number of assumptions, and such labels act as assumptions. For future work, an improvement is possible. These

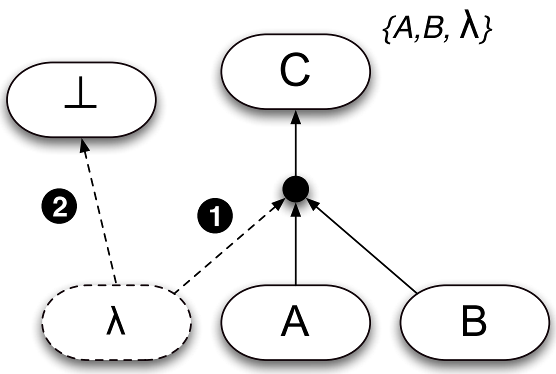


Figure 7.1: Simulating retraction with a 'dummy' node λ

'dummy' labels are not assumptions in the full sense: there is no need to minimize their occurrence or eliminate duplicate labels in explanations; also, such labels do not participate in inconsistencies. If the ATMS implementation is therefore modified to split explanations into two sets: one with tasks and one with rule labels, the operations on the second sets (actually bags) are much cheaper.

The problem solver is also responsible for the ASSERT-ATTITUDE operator. This is simply done using a look-up table with node-names, and associating an attitude (preferred or mandatory) with each. A similar approach applies for preference relations. The problem solver is also responsible for calculating the distance functions (covered in §8.3.1).

7.2 Deciding on a Solution

After ASKing questions of the REKB, our second design task is to choose a particular solution from those returned. We will say a set of tasks S is a solution if S is a set returned by MINIMAL-GOAL-ACHIEVEMENT over the mandatory goals. Then by definition, for non-empty S, S is minimal and consistent and achieves the mandatory goals. Since all S satisfy our 'must-have' requirements, each is equally valid. To choose between them involves the use of a decision procedure. There seem to be six obvious decision criteria:

1. We can select a random S.

2. We can choose to minimize the size of S in terms of tasks which must be implemented.

3. If the data is available, we can choose the S with the lowest cost, for some numeric cost function *f(S)*. For example, we could assign each task a cost to implement.

4. We may want to maximize the preferred atoms S entails.

5. Techne allows us to specify binary preference relations between atoms in the REKB, so we might choose the S which satisfies the most preference relations.

6. We could define a decision rule that assigned weights to these preceding criteria and maximized their sum.

Let us turn again to our payment card case study to illustrate this. Earlier I identified alternative means of satisfying compliance tests (§5.4). One example was supporting the PCI-DSS requirement that server functionality be separated (so that root access on one server does not mean root access on the others). One alternative is *A1*: implement virtualized servers; the other is *A2*: purchase separate machines per server role. We must achieve either of these two alternatives in order to satisfy the goal in the PCI-DSS. We say that these alternatives structure the solution search space: there are now two potential solutions ($A1 \in S_1$ OR $A2 \in S_2$).

Assuming that both alternatives produce candidate solutions (that is, both are entailed by an S which satisfies the mandatory elements), we need some way to decide between *virtualization* and *purchasing servers*. To do this, we turn to our decision criteria. For (1), we can merely 'flip a coin' to choose. For (2), we select the smallest cardinality S. For (3), we must be given cost values. For example, we might be told that it will take three weeks to order servers, and two weeks to virtualize existing servers. For (4), we might have added the *preferred* property onto implement virtualized servers, and thereby select that alternative; for (5), on the other hand, we might have added a preference relation for *A2* over *A1*, indicating we favor the solution with *A2*. If we combine these criteria (criteria 6, e.g., look to maximize the inclusion of preferred goals AND maximize the satisfaction of preference relations) we must do multi-objective optimization, as these are not comparable criteria. Choosing between multiple solutions which have multi-dimensional criteria is multi-objective optimization.

Multi-objective optimization is the problem of deciding between incomparables and maximizing the satisfaction of both. Consider the case of choosing a hotel for vacation [3]. The traveller would like a cheap room, but at the same time, would like to walk to the beach in a short amount of time. However, for his destination there is no *single* hotel which is simultaneously the cheapest and the closest to the beach. Instead, the available hotels can be visualized as satisfying two dimensions of preferences, as shown in Fig. 7.2. The horizontal axis represents the price of a hotel; the vertical axis the distance to the beach. Point A represents a hotel which is farther from the beach but cheap, while Point B represents a hotel which is conversely pricier but closer to the beach. Point C is a hotel which we will say is *dominated* by both A and B: it is neither closer to the beach than A, nor cheaper than B. The

[3]This example is derived from [Kossmann et al., 2001].

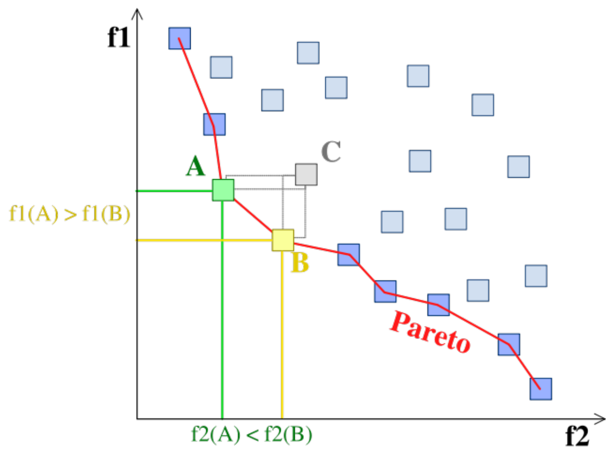


Figure 7.2: Sample Pareto front. ©Johann Dréo, used under Gnu FDL.

red line is called the Pareto-front: it represents dominant choices. The traveller should be equally happy with any solution that is on the Pareto-front; furthermore, we cannot make any more refinements of the problem without more information from the traveller (e.g., that he or she would rather pay more than be close to the beach).

The set of solutions to the requirements problem similarly fall along a Pareto-front. In this chapter the dimensions are a) number of preferred requirements satisfied and b) number of preference relations satisfied.

7.2.1 First Approach: pQGM

In [Ernst et al., 2010], my co-authors and I looked at selecting solutions to the requirements problem using a slightly different formulation of the requirements problem. The essence, however, is the same: a goal model and a reasoning procedure that identifies candidate solutions. In this section, I will outline how this process is then used to find a particular solution, using the multi-objective problem previously mentioned: to maximize both the number of preferred goals satisfied and the number of preference relations satisfied.

Background: The ESP Case Study

Figures 7.3 and 7.4 present a subset of the case study I created for this section. The solution selection techniques will be used to model a real life setting concerning enterprise web portal development (ESP). I base the scenario on the documentation that the provincial Ministry of Government Services of Ontario (Canada) (MGS) provides, specifically on the description of the content management requirements. This is a series of reports from a 2003 project to re-work the provincial government's enterprise-wide citizen

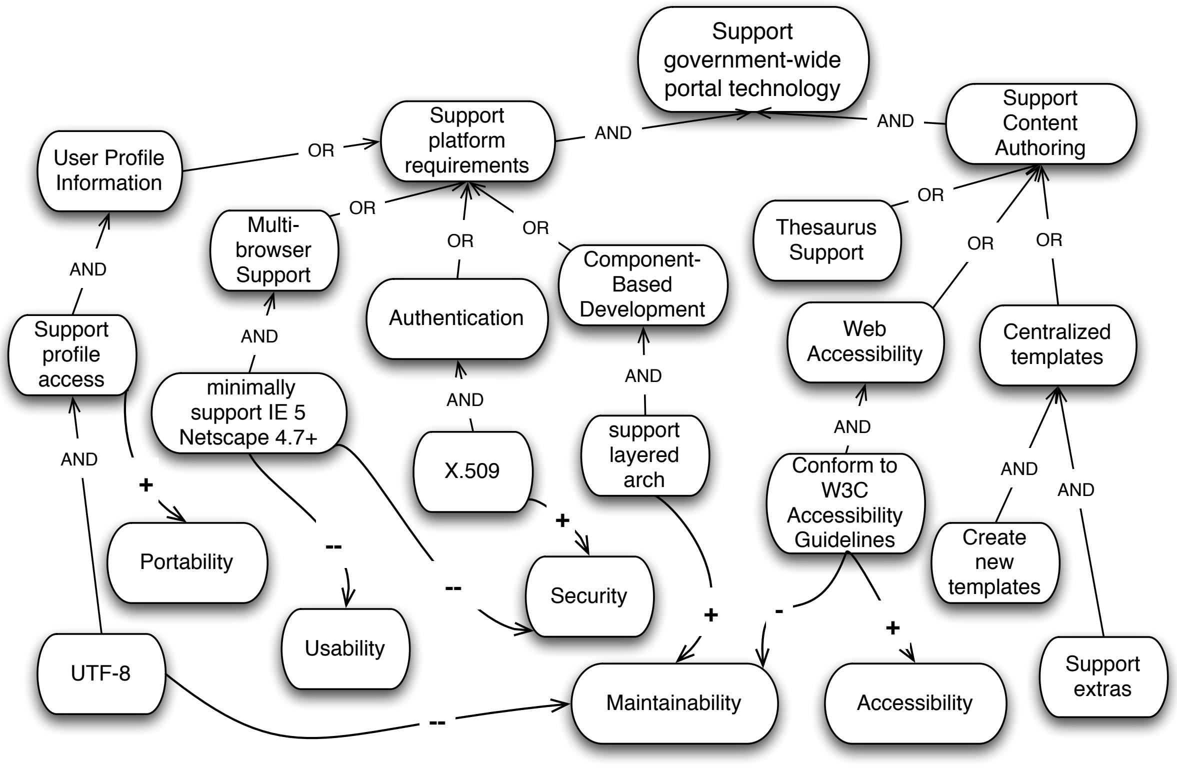


Figure 7.3: A partial QGM model of the case study.

access portals, as part of an e-Government initiative. The content management requirements detail the specific technical aspects such a portal would have to meet[4]. The extended model contains 231 goals and several hundred relationships. This is the same size as real-world problems modeled with KAOS [van Lamsweerde, 2008, p. 248].

Prioritized Qualitative Goal Models

The major change in this section is that I use an early approach to goal models as the 'input' to the solution selection algorithms. I leverage the qualitative goal modeling framework, henceforth 'QGM', as defined in Sebastiani et al. [2004]. That paper defines an axiomatization of goal models that transforms the goal labeling problem into a boolean satisfiability (SAT) problem (leveraging the power of off-the-shelf SAT solvers). The goal labeling problem, as stated in that paper, is "to know if there is a label assignment for leaf nodes of a goal graph that satisfies/denies all root goals [Sebastiani et al., 2004, p. 21]". This approach can be used, among others, to find what top-level goals can be achieved by some set of leaf goals/tasks ("forward" or ARE-GOALS-ACHIEVED-FROM), or conversely, given some top-level

[4]available at http://www.mgs.gov.on.ca/en/IAndIT/158445.html

goals which are mandated to be True, what combination of tasks will achieve them ("backward" or MINIMAL-GOAL-ACHIEVEMENT). I will consider the SAT approach to REKB operations in §7.5.

The connection to the REKB approach is as a preliminary approach to a notion of a requirements database, and the operations ARE-GOALS-ACHIEVED-FROM and MINIMAL-GOAL-ACHIEVEMENT. In this case, I use repeated calls to a SAT solver to identify all candidate solutions, and the language $\mathcal{L}$ is different.

In any case, a solution to the requirements problem amounts to the same definition as we have looked at previously: given the inputs, which in this case are goals, can we satisfy the mandatory goals? The mapping to Techne is reasonably straightforward, if not explicitly detailed: rather than tasks, we have input goals, that is, goals which have been ASSUMED true. The ontological distinction between a "leaf" goal and a task, while important, does not impact the reasoning procedures.

The major important difference with Techne is what constitutes satisfaction of a goal. As I show below, in the QGM framework that amounts to a four-valued logic approach, an approach we first tried with Techne Jureta et al. [2009a] but ultimately did not find useful. QGM's approach forces the treatment of inconsistency to be dealt with outside of the reasoning process, as we will see. The limited language for Techne used in my thesis is less expressive in the sense that it cannot cope with QGM's notion of partially satisfying a goal.

Let us briefly recapitulate the definitions introduced in Sebastiani et al. [2004]. There are sets of goal nodes G_i, and relations $R_i \subseteq \wp G \times G$.

Relations have sorts *Decomposition*: $\{and,\ or\}$, which are $(n+1)$-ary relations; and *Contributions* $\{$*+s,++s,+d,++d,-s,--s,-d,--d,++,+,--,-*$\}$, which are binary. A goal graph is a pair $\langle \mathcal{G}, \mathcal{R} \rangle$ where $\mathcal{G}$ is a set of goal nodes and $\mathcal{R}$ is a set of goal relations.

Truth predicates are introduced, representing the degree of evidence for the satisfaction of a goal: $PS(G), FS(G), PD(G), FD(G)$, denoting (F)ull or (P)artial evidence of (S)atisfaction or (D)enial for G. A total order on satisfaction (resp. denial) is introduced as $FS(G) \geq PS(G) \geq \top$ (no evidence). In this use of $\top$, it is indicating the trivially true statement that there is at least null evidence for G being satisfied (per Giorgini et al. [2003]. This permits an axiomatization of this graphical model into propositional logic, so that the statement "A contributes some positive evidence to the satisfaction of B", represented $A \overset{+s}{\longmapsto} B$ becomes the logical axiom $PS(a) \rightarrow PS(b)$. The complete axiomatization from which this section is derived is available in Sebastiani et al. [2004].

With respect to the ATMS formulation of the requirements problem presented earlier, it should be possible to recreate the entire set of logical formulas of [Sebastiani et al., 2004] inside the ATMS: for each goal g, the problem solver can create nodes FS_g, PS_g, PD_g, FD_g, which simulate the predicates introduced before. For relations such as $++s$ these nodes are joined according to the label propagation rules, i.e., if we have the Sebastiani et al. relation $A \xmapsto{+s} B$, the ATMS creates the justification $PS_A \rightarrow PS_B$. The additional axioms which ensure completion semantics are not necessary, nor is the use of the MinSAT reasoning tool, since we can leverage ATMS to find the minimal explanations.

Extra-logical extensions. To manage preferences and preferred elements, I added extra-logical elements to QGM. These elements do not affect the evaluation of candidate solutions (i.e., whether there is a satisfying assignment), thus the use of the term "extra-logical". This extension is called *prioritized* Qualitative Goal Models (pQGM). Models in this language form requirements nets (*r-nets*). pQGM *r-nets* allow us to capture stakeholder attitudes on elements in the model. Attitudes (i.e., emotions, feelings, and moods) are captured via the Techne notions of preferred goals and preference relations.

Optionality is an attribute of any concept indicating its preferred or mandatory status. Being *mandatory* means that the element e in question must be fully satisfied in any solution (in the QGM formalization, $FS(e)$ must be true) and not partially denied ($PD(e)$ must be false). (In general, element e will be said to be "satisfied" by a solution/labeling iff $FS(e) \wedge \neg PD(e)$ is true.) Being *preferred* means that, although a solution does not have to satisfy this goal in order to be acceptable, solutions that do satisfy it are more desirable than those which do not, all else being equal.

We consider non-functional requirements (NFRs) like *Usability* to be ideal candidates for being considered preferred goals: if not achieved, the system still functions, but achievement is desirable if possible.

Alternatives arise when there is more than one possible solution of the problem (i.e., more than one possible satisfying assignment). In goal models with decomposition, this typically occurs when there is an OR-decomposition. We treat each branch of the OR-decomposition as a separate alternative solution to the requirements problem. Note that even for very small numbers of OR-decompositions one generates combinatorially many alternatives.

Preferences are identical to the corresponding Techne notion: binary relationships defined between individual elements in an *r-net*. A preference relationship compares concepts in terms of desirability. A preference from a goal to another goal in a pQGM *r-net* indicates that the former is strictly preferred to the latter. Preferences are used to select between alternatives.

Identifying Candidate Solutions

The purpose of creating a pQGM model is to use it to find solutions to the requirements problem it defines. Finding a solution is to identify candidate solutions, and to then select from the set of candidates. To do so, I identify admissible models ('possible solutions'); satisfy as many preferred requirements as possible; identify alternatives; then filter the set of alternatives using user-expressed preferences.

What are the questions that pQGM can answer? We want to know whether, for the mandatory goals defined – typically the top-level goals as these are most general – there is some set of input nodes which can satisfy them. Furthermore, we want these sets of inputs to be strictly preferred to other possible sets, and include or result in as many preferred goals as possible. This notion of maximizing the number of satisfied preferred elements is the approach taken here: in other cases, such as where we are given weights for each goal that is preferred, another criteria (such as minimizing cost) might be better.

I show some answers to these questions in §7.4.1, below. With respect to the pQGM model shown in Fig. 7.4, a dominant solution (stippled nodes) consists of the goals *Support government-wide portal technology, Support Content Authoring, Web Accessibility, Conform to W3C Accessibility Guidelines, Support platform requirements, Authentication, X.509, User Profile Information, Support profile access, UTF-8, Accessibility, Security, Portability.* Goals such as *Minimally support IE5* are alternatives that are dominated, and therefore not included (in this case, because they break the *Usability* and *Security* preferred goals). Deriving this is the focus of this section.

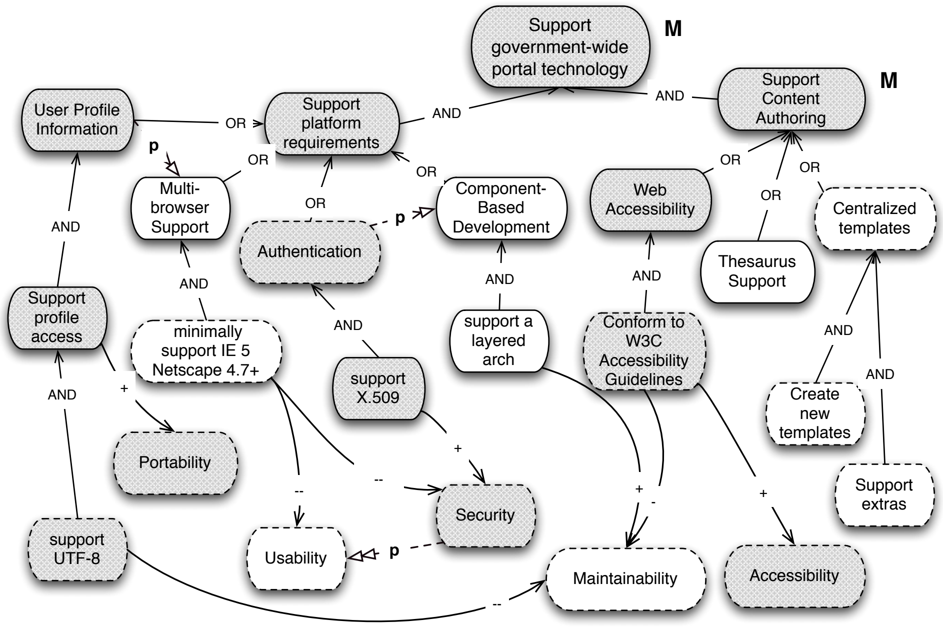


Figure 7.4: An example pQGM *r-net*. Preferred goals have dashed outlines and mandatory elements are labeled 'M'. Preferences are represented with 'p' arcs on double-headed arrows. $\bar{R}$ consists of the elements outlined in solid borders and $\bar{S}$ the stippled elements (a candidate solution).

Identifying admissible models – The first step is to find satisfying label assignments to the model elements. I rely on the solution to the backwards propagation problem of Sebastiani et al. [2004]. We take an attitude-free *r-net* $\bar{R}$ (*attitude-free* meaning one without preferred elements, edges leading to/from them, and without considering preferences), and label the model. The labeling procedure encodes the user's desired output values (Φ_{outval}), the model configuration (Φ_{graph}) and the backwards search axiomatization $\Phi_{backward}$ (cf. [Sebastiani et al., 2004, p. 8]). In the case of nodes e representing "hard" goals and especially tasks, one might require only truth assignments that satisfy axioms forcing binary "holds"/"does not hold" of e, through axioms $FS(e) \rightarrow \neg FD(e)$ and $FS(e) \vee FN(e)$, forming additional sets $\Phi_{conflict}$ and $\Phi_{constrain}$ in Sebastiani et al. [2004]. The output of this is a satisfying truth assignment μ, or *None*, if there is no assignment that makes the mandatory nodes $FS \wedge \neg PD$.

If there was a satisfying assignment, then the attitude-free model $\bar{R}$ is admissible (that is, it forms a candidate solution), as the mandatory nodes were satisfied, and call the resulting labeled goal model $\bar{R}_m$. At this point, it suffices to identify that there is admissibility. Subsequent stages handle alternatives.

Identifying Preferred Elements

I now turn to consideration of the preferred goals of the initial *r-net R*. In the diagrams, a node is marked preferred by having dashed outlines. (This could of course be formalized by using a meta-predicate **O**.) Although a solution does not have to satisfy such a goal in order to be admissible, solutions that do satisfy it are more desirable than ones that do not, all else being equal. For example, in Fig. 7.4, we would prefer to satisfy all the dashed nodes (*Conform to W3C accessibility guidelines*, etc.). However, the use of preferred goals permits the acceptance of solutions which only satisfy *some* of these nodes.

Consider the example in Fig. 7.4. In this example we have an NFR *Usability* and incoming concepts which can satisfy the goal. By specification, the algorithm should find the maximal sets of preferred elements that can be added to the mandatory ones while preserving admissibility. It is important to note that in our framework preferred elements can interact. This means that adding a preferred goal to R_m could render a) the new model inadmissible b) previously admissible preferred goals inadmissible. This reflects the notion that introducing an preferred goal has consequences.

Preferred element identification. I describe the naïve implementation in Algorithm 6. To begin there is an admissible r-net, R_m, e.g., a set of mandatory goals, along with a map, $\mathcal{T}$, from elements of R_m to the set of label assignments for R_m, e.g, $\{PD, PS, FD, FS\}$. The algorithm is given a set, $\mathcal{O}$, the set of preferred elements, with $\mathcal{O} \cup R_m = R$. In our example, a subset of $\mathcal{O}$ is equal to the set of goals culminating in *Usability*.

For each subset *input* of $\mathcal{O}$ (i.e., element of $\wp(\mathcal{O})$), with the exception of the empty set, add it to $\bar{R}_m$, the admissible solution. I then use the SAT solver to find a satisfying assignment, checking for admissibility. If there is an admissible solution (a set O such that $O \cup R_m$ is admissible), it can be added to the set of acceptable solutions, $\mathcal{S}_a$. Now, because of the order in which we traverse subsets, this solution is a maximal set, and the proper subsets of O contain fewer preferred goals. I therefore remove these sets from the collection being traversed.

The running time of this naive approach is clearly impractical since in the worst-case, it must check all 2^n subsets of the set $\wp(\mathcal{O})$, where n is the number of preferred elements. (The worst-case could arise if all preferred goals invalidate all other preferred goals.) This is on top of the complexity of each SAT test! Furthermore, the result set might be quite large. I will show some mechanisms to improve this using, among other things, a local search technique in §7.4.1.

Algorithm 6: NAIVE-SELECT

```
  Input: A solution $R_m$ and a set of preferred goals, $\mathcal{O}$
  Output: All maximal admissible sets of preferred goals that can be added to $R_m$
  $\mathcal{O}_m := \{\}$
  foreach input $\in \wp(\mathcal{O})$ do
      // in a traversal of sets in non-increasing size
      if admissible(input $\cup R_m$ then
          $\mathcal{O}_m := \mathcal{O}_m \cup$ input
          Remove subsets of input from $\mathcal{O}$
      end
  end
  return $\mathcal{O}_m$
```

Selecting Alternative Solutions

The input into the penultimate phase consists of the admissible, labeled *r-net* R_m, along with a set, possibly empty, of sets of preferred elements that can be joined to that *r-net*, $\mathcal{O}$. The number of current solutions, then, is the size of $\mathcal{O}+1$.

The goal of this phase is to identify alternatives that are created at disjunctions in the model. I do this by converting each admissible solution $s \in S : R_m \cup\ O \in \mathcal{O}$ to a boolean formula (a traversal of the AND-OR graph), and then converting this formula to conjunctive normal form. This is the accepted format for satisfiability checking (SAT). I pass this representation of the *r-net* as the SAT formula Φ to a SAT solver, store, then negate the resulting satisfiability model μ and add it as a conjunct to Φ. I repeat this process until the result is no longer satisfiable (enumerating all satisfying assignments). This produces a set of possible alternative solutions, e.g. $\mu_i, \mu_{i+1}, .., \mu_n$. This is converted to a set of sets of concepts that are solutions, $\mathcal{S}_a$.

Solution selection Our final step is to prune the sets of solution *r-nets*, $\mathcal{S}_a$, using stakeholder valuations over individual elements – expressed as preferences and (possibly) costs. We do not prune before finding preferred goals since we might discard a dominated set in favour of one that is ultimately inadmissible. Similarly, we do not risk the possibility of discarding a set of preferred elements that is nonetheless strictly preferred to another set, since by definition, if set O' is admissible and yet not selected, there was a set $O \supset O'$ that contains the same elements (and therefore preferences) and was admissible.

The following case may arise: consider admissible R-net with nodes {1,2,3}. Let 1 and 2 be disjunct inferences for 3. Therefore potential solutions include {1,3} and {2,3}. In the case where 2 is preferred to 1, we would select {2,3} as the solution that dominates. However, if we haven't yet considered preferred goals, we may miss a situation in which another node, 4, is attached to node 1. Thus even though we prefer node 2 to node 1, we shouldn't remove from the user the opportunity to consider the solution {1,3,4} (which is not dominated).

Selection using preferences. We will use in the text the notation pq if the r-net indicates that node p is preferred to node q, and let $\geq$ be the transitive reflexive closure of .

We use this to define the function *Dominate(M,N)*, mapping $\mathcal{S}$ *x* $\mathcal{S}$ to booleans as follows:

$$Dominate(M, N) = \forall n \in N.\exists m \in M : m \geq n$$

Intuitively, every element of a dominated set is equal to a value in the dominant one, or is (transitively) less preferred, as I covered in §7.2. Note that the *Dominates* relation is a partial order, and I can therefore choose maximal elements according to it.

Selection using cost. Although not shown in the case study, pQGN provides for solution selection using a simple cost operator, whenever such numbers are available. Clearly there are many cost dimensions to consider. For a given cost function, I define a *min_cost(value, increment,set)* function which ranks the proposed solutions using the cost of the solution as a total ordering. The meaning of *value* is as an upper cost threshold ('return all solutions below *value*'), and *increment* as a relaxation step in the case where no solutions fall under the threshold. Note that we are ranking admissible solutions, and not just requirements.

With pQGM, I introduced an extension to a well-known formal goal reasoning procedure. This extension allowed for the comparison of solutions to the requirements problem using preferences and preferred goals. I described some techniques for generating solution alternatives and comparing them. While the non Techne QGM framework is at the heart of this technique, the stages for selecting solutions using preferred elements and preference relations is common to both. In that sense, the REKB questions MINIMAL-GOAL-ACHIEVEMENT and ARE-GOALS-ACHIEVED-FROM, as well as QGM's use of SAT and MAXSAT, are merely inputs for the decision procedure.

7.2.2 Selecting Solutions

There is clearly more room for research in the area of solution selection. My approach of using a combination of satisfied preference relations and inclusion of preferred elements is one way to do this, but there are others, of course. For example, the work of Durillo et al. [2010] on the Next Release Problem has illustrated a numeric optimization approach. This, however, presupposes the existence of meaningful and reliable numeric weights, which are often not available for the type of design problems at which I have been looking.

7.3 Tool support

Let us turn to somewhat more mundane matters (but no less important for all that it is mundane). This section outlines tools for supporting the operations and implementations previously described. I begin by discussing how users (be they stakeholders or analysts) can interact with the REKB and the problem solver. This includes a domain-specific language for Techne, and a graphical editor for manipulating goal models.

7.3.1 DSLs

A domain-specific language (DSL) is sometimes called a “little language”. DSLs allow users to express their problem using domain terminology they are familiar with. A compiler/transformer then converts this DSL syntax into a more traditional language (LISP, in this case), and eventually into machine instructions.

I have created a simple DSL for Techne, which allows us to express requirements problems using Techne concepts like domain assumption, goal, and justification.

Listing 1 gives an example of the DSL syntax for capturing requirements problems in Techne.[5]

[5]The complete model and source code is available at http://github.com/neilernst Techne-TMS.

```
(defvar
g0 (declare-atomic nil "Comply with PCI DSS" :GOAL *rekb*)
g1 (declare-atomic nil "Build and Maintain Secure Network" :GOAL *rekb*)
g2 (declare-atomic nil "Protect Cardholder Data" :GOAL *rekb*)
g3 (declare-atomic nil "Maintain a Vulnerability Mgmt Program" :GOAL *rekb*)
g4 (declare-atomic nil "Implement Strong Access Control Measures" :GOAL *rekb*)
g5 (declare-atomic nil "Regularly Monitor and Test Networks" :GOAL *rekb*)
g6 (declare-atomic nil "Maintain an Information Security Policy" :GOAL *rekb*))

(assert-formula  gc1.2.1.2.1 (list g0) :DA *rekb*)
```

Listing 1: Partial representation of PCI case study in Techne

The DSL could be improved to remove the need to add variable definitions and some of the brackets. However, one is able to see a straightforward mapping between the listing and the requirements problem from Fig. 2.1.

7.3.2 Graphical Editor

Another approach, aside from the textual simplicity of the DSL, is to define a visual editor for requirements problems. This necessarily involves defining a visual syntax for Techne, which to date has not been a research focus. The challenge, as with any visual language, is to define a notation that is intuitive and yet scalable. Fig. 2.1, for example, is a very simplistic approach to a visual formalism. A more sophisticated approach was shown in Fig. 5.1, but it is not one that was tested with users.

In Ernst et al. [2008] we presented a more sophisticated visual editor that worked with OpenOME goal models, which are based on the semantics of Sebastiani et al. [2004]. OpenOME[6] is an Eclipse-based goal modeling tool supporting several languages, including Telos [Mylopoulos et al., 1990] and q7 [Yu et al., 2005a].

OpenOME has three flavours of editor to work with goal models:

- A text editor for creating q7 files, supporting syntax highlighting and checking (generated from its domain-specific grammar using xtext in the openArchitectureWare model-driven software development suite);
- An object-oriented editor of goal models with features of tree-based outline of objects and table-based field property views (generated from the Eclipse Modeling Framework (EMF));
- A graph editor of the goal models. This combines the features of an EMF editor plus graphical ones such as graph layout, zooming, tool panels, printing, etc. This editor is generated from the

[6] https://se.cs.toronto.edu/trac/ome

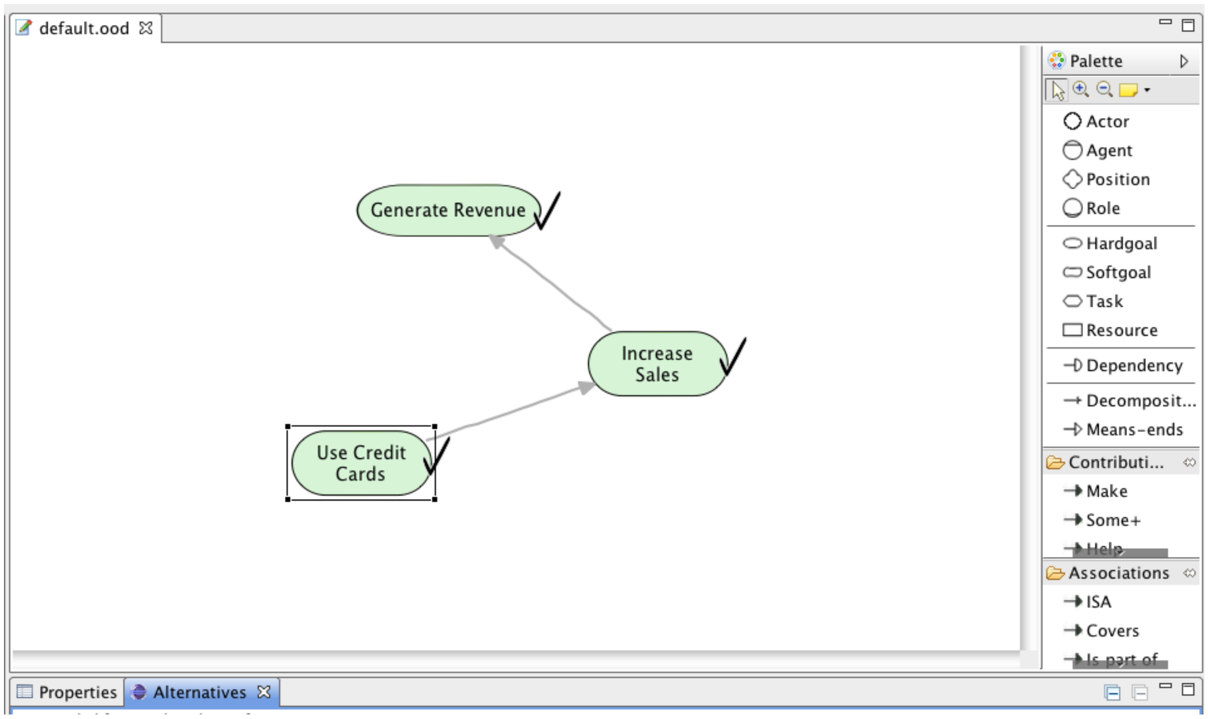


Figure 7.5: The OpenOME user interface.

Graphical Modeling Framework (GMF). With my colleague Yijun Yu, I created and built this portion of OpenOME for working with goal models.

A screen capture of the OpenOME user interface is shown in 7.5. It is future work to update OpenOME to work with the current form of Techne.

7.4 Implementation Feasibility

Now that we have looked at the implementation of the REKB operations I turn to examining the feasibility of those implementations.

The goal of the evaluation phase is two-fold. Recall Wieringa et al. [2006]'s activities in the engineering cycle from §1.3. The preceding sections discussed **solution implementation**. The subsequent phase, which was out of scope for my thesis, is **implementation evaluation** which can only be done *in situ*: by evaluating the use of the REKB approach in industrial settings. What I discuss below is rather feasibility evaluation, which should be performed prior to releasing code to industry. Feasibility studies examine the practicability of the approach, and are concerned with verifying that the implementation adheres to the design specification. Feasibility also tests whether the approach will work in practice, particularly with respect to usability and scalability.

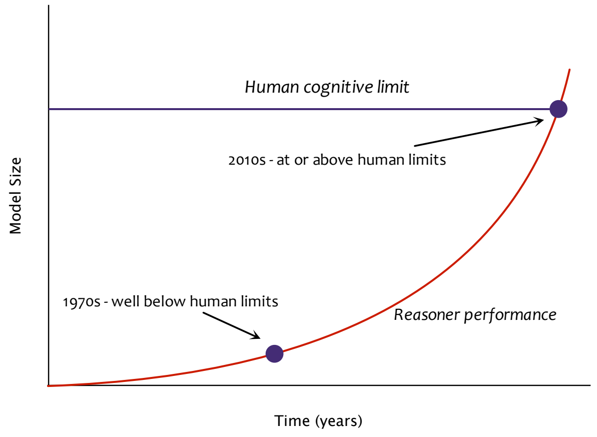


Figure 7.6: Human cognitive limits and automated reasoning on requirements models.

Figure 7.6 shows a simplified chart comparing human cognitive performance to reasoner performance on a constant-size requirements model over time. The notion is simple: human cognitive performance is invariant (more or less): empirical studies have reported that humans cannot manage models larger than 200–300 nodes and associations (and conceivably much less, for non-problem specialists). The problem for automated reasoners was that when first conceptualized (like Telos), they could not manage models of even this complexity.

The current performance of the reasoners, even for reasoning tasks which are NP-complete such as abduction and satisfiability, is sufficient to manage models of this complexity. This is largely because of two factors. One is the ongoing work on the underlying reasoning engines, like SAT solvers; the other is the computational improvements due to Moore's law. Compare the reasoning done in Menzies [1996] with that done in Ernst et al. [2011]. In both cases the reasoning is abductive and performed on requirements models; in 1996 Menzies was conducting experiments on a Powerbook 170, a machine with 4 MB of RAM as standard, and a clock cycle of 25 MHz. Comparatively, the machine most of the experiments conducted in this thesis has 4 GB of RAM, with a 2.5 GHz processor.

Menzies [1996] surveys the practical sizes of some contemporary expert systems: he claims a range of $55 \leq \mathcal{V} \leq 510$ for vertices and $2 \leq \frac{\mathcal{E}}{\mathcal{V}} \leq 7$ for the complexity of the edge-ratio.

For models of so-called early requirements, which are concerned with understanding design tradeoffs and goal allocations to agents, models tend to be smaller, since they are often used for direct interaction with the clients. For example, in the industrial case study described in Easterbrook et al. [2005], the models ranged in size from 100-200 nodes and were very tight on scope. Reports from the study indicated that even at these sizes, models were very difficult to work with. Industrial experience with the KAOS methodology reports model sizes that were, on average, 540 goals and requirements [van

Lamsweerde, 2004]. Early requirements models are complex, and more decision- and analysis-oriented than specification models. Working on models beyond this range requires modularization and separation of concerns, as argued in Fuxman et al. [2004]. It would be useful to define some sample models and model metrics for comparison purposes. Models should be characterized in terms of overall size, branching complexity (e.g. out-degree), number of alternatives, number of inputs, and so on.

These results apply only to models that humans are expected to comprehend. This typically means early-phase requirements models where design is the central activity. If humans do not need to understand the entire model, or if the model is used to do code-generation, for example, it could get considerably larger. This is the case in e.g. terminology services like the National Cancer Institute's Enterprise Vocabulary Service, which is thousands of concepts in size. At this point, consensus is that local search strategies are necessary for reasoning on these models.

Let us first define what constitutes 'reasonable performance'. Achieving consensus on this question is a perennial problem in requirements research. Models mean different things to different people, and there is a tension between scaling to models with thousands of elements that are used in late-phase RE, for example automotive product lines, and models that are used in academic settings or intended for design and decision support. A model represented in DOORS, for example, might consist of one thousand requirements, but very limited connectivity, and no notion of alternatives. An i* model, on the other hand, is a much denser information space, with many interconnections and complex notions for refinement and satisfaction.

My position is that a formal reasoning tool must handle *early* requirements models with an upper size ranging in the hundreds of nodes. One reason for this limit is that comprehension of such large models is very difficult. For example, in the industrial case study described in Easterbrook et al. [2005], the models ranged in size from 100-200 nodes and were very tight on scope. Reports from the study indicated that even at these sizes, models were very difficult to work with. Industrial experience with the KAOS methodology reports model sizes that were, on average, 540 goals and requirements van Lamsweerde [2004]. Early requirements models are complex, and more decision- and analysis-oriented than specification models. Working on models beyond this range requires modularization and separation of concerns, as argued in Fuxman et al. [2004]. It would be useful to define some sample models and model metrics for comparison purposes. Models should be characterized in terms of overall size, branching complexity (e.g. out-degree), number of alternatives, number of inputs, and so on.

The other limit we are concerned with is human task performance. A widely-used rule of thumb is the 0.1/1/10 second rule [Miller, 1968]. In brief, the rule states that 0.1 seconds is the limit for users to feel that they are directly manipulating something; 1 second is the limit for users to feel as though they aren't waiting for the computer; and 10 seconds is the upper limit on people's willingness to wait for a task to finish. After ten seconds, they will turn to something else.

If we want to support the use of the REKB as an interactive design aide, it needs to provide responses to user queries (ASK operations) at most under ten seconds, and preferably under 1 second. Ideally, these answers would be delivered in milliseconds, and allow the user to manipulate model elements in real-time, and immediately see the consequences of his or her decisions.

This section has outlined two criteria we must consider when designing support for interactive requirements decision-making. The first consideration must be the size of the problem. The second consideration is the speed of the reasoning: it must return results within ten seconds in order to support interactivity.

This section examines feasibility of both the naive approach, detailed above, and also discusses some simple improvements that can greatly improve the reasoning performance on larger models. We can draw an analogy to the performance of reasoners on the Boolean satisfiability problem (SAT). All algorithms are exponential, since the problem is NP-complete. However, with clever approaches such as the DPLL algorithm's unit propagation [Davis et al., 1962], clause learning, and smart backtracking, the average SAT problem can be quickly conquered, although some problems remain difficult.

Therefore the goal of optimizing the reasoning techniques for the requirements problem is likewise to improve average-case performance. The work of [Mitchell et al., 1992] has showed that there is a threshold for SAT problems, based on the clause/variable ratio, past which problems get dramatically harder to solve. It seems reasonable to think that the same is true of requirements problems, which can be reduced to SAT instances.

The RE literature contains several techniques to simplify reasoning on RE models, many of which focus on partitioning the model into smaller pieces based on various criteria. The caveat here is that the merging process can be complex due to ontological drift and designation clashes [Easterbrook et al., 2005].

- Stakeholder views can be used to separate the problem into fragments which are relevant to a particular person.
- Gotel and Finkelstein [1995]'s *contribution structures* can associate certain parts of the model with a particular agent.
- Root requirements analysis [Robinson and Pawlowski, 1998] can be used to partition the model using the top-level (more abstract) requirements. The challenge is handling the potential interactions.

- Aspects as described in Yu et al. [2004] are cross-cutting concerns such as non-functional requirements. Focusing on a particular aspect excludes non-relevant portions of the model.
- Wang and Mylopoulos [2009] described a process similar to unit resolution in the DPLL algorithm for collapsing multiple requirements into a single representative based on AND-subtrees.

In the following sections I evaluate the feasibility of the various operations for the static requirements problem. I begin with the scalability of the pQGM approach and the local search technique I used to find solutions. The remainder of this section looks at the feasibility of the ATMS approach to the REKB operation MINIMAL-GOAL-ACHIEVEMENT, evaluating it on random models and the case study. I conclude by suggesting some simple improvements.

7.4.1 Optimizations for pQGM

The naive pQGM approach has running time exponential in the number of preferred goals. There are several improvements we might consider.

Pruning the set of preferred goals. I introduce two ways to reduce the number of option sets the naive algorithm finds, in order to reduce problem size. One approach prunes the initial sets of preferred goals, and another reduces the admissible option set presented to the user. The intuition is to support a form of restriction on database queries; some requirements might be useful in some scenarios but not others, yet we would like to retain them in the original model. For example, I might implement the ESP model in an environment which does not need authentication.

The first approach defines simple operations over preferred goals – operators on $\wp(\mathcal{O})$ that act as constraints. Each element (a set) in $\wp(\mathcal{O})$ has two variables, cardinality and membership. I allow constraints on option set cardinality (boolean inequalities), and option set membership (inclusion or exclusion). An equivalent expression in SQL might be `DELETE FROM options WHERE Size <n` and `DELETE FROM options WHERE $x IN options`.

The second approach is to make use of pQGM's preference relations to remove option sets which are dominated by other sets. In our example, with $P = \{(\textit{Authentication}, \textit{Component-based development}), (\textit{Security}, \textit{Usability})\}$, if we found admissible solutions S_1 containing $\{\textit{Security}, \textit{Authentication}\}$, and S_2 containing $\{(\textit{Component-based development})\}$, the algorithm would discard S_2 in the case where both are admissible. This makes use of the Dominate function defined in §7.2.1.

Local search. The naive approach must search through each member of the set of possible options, an exponential worst-case running time. This is clearly infeasible. A more tractable approach is to define a local search algorithm to find, in a bounded amount of time, a locally optimal solution.

The objective is to discover solutions to requirements problems, that is, configurations of tasks which satisfy the requirements, constrained by the domain assumptions. A solution to this problem must satisfy the *mandatory* goals, and do as well as possible with respect to the *preferred* goals. This is a search problem: the problem space is provided, which is the various configurations of tasks. The problem is to determine which sets of tasks are optimal with respect to the criteria. Note that the complexity of this problem appears to be exponential in the number of tasks, since one must consider the powerset of tasks $\wp(T)$, whose cardinality is $2^n - 1$.

A naive approach would search the entire space, considering each subset of tasks. However, this is costly in space and in time if the number of tasks is large. A **local search** approach tries to incrementally improve the solution, moving from one set to another in the local neighborhood. Here, the path cost (the route taken) is irrelevant, but the search cost is paramount (the time and memory taken to find the solution).

A simple local search technique is hill-climbing. If the "top of the hill" is the globally optimal solution, a hill-climber will begin with a single task and iteratively add more tasks until the requirements problem is solved. If adding another task either fails to satisfy the mandatory goals (perhaps because it conflicts with the existing selection of tasks), or does not do better with respect to preferred elements, then the hill-climber is finished. The problem is that this technique will fail if no solution is found, or fail to advance beyond this local maximum (e.g., the solution found is a set of tasks that satisfy the mandatory goals, but could do better in maximizing the satisfied options).

The solution to this is to restart the search, to in effect move the search away from the plateau it has reached. There are a number of ways to do this, but the typical approach is to randomly restart at another point in the search space, and repeat the procedure. Some form of timer is kept in order to limit the number of restarts, and there are several techniques for choosing efficient ways to restart.

Algorithm 7: TABU-SEARCH

```
Input: solution $R_m$, set of options $\mathcal{O}$, time limit $tlim$, tabu expiration $expire$
Output: A locally optimal set of optionsets $\mathcal{O}_t$
// initialize variables here
while $time < tlim$ and $candidate \notin tabu\_list$ do
    $\mathcal{O}_t = \mathcal{O}_t$ + Tabu-Move($\emptyset$, $\mathcal{O}$, $\mathcal{O}_t$, tabu, time)
    if $time \% expire \leq 1$ then
        tabu $\leftarrow \emptyset$
    end
end
foreach $o', o'' \in \mathcal{O}_t$ do
    if $o' \subset o''$ then
        $\mathcal{O}_t$ - $o$
    end
end
```

For pQGM, I implemented Tabu search [Glover, 1986], captured in Algorithms 7 and 8. The algorithm

Algorithm 8: TABU-MOVE

```
Input: candidate, remainder, solution, tabu, time
Output: solution
// initialize variables
step = 0
while $step < radius$ do
    tmp = candidate
    selected = Random.Choice(remainder)
    tmp = tmp + selected
    step = step + 1
    if $selected \notin tabu\_list$ and $tmp \notin solution$ then
        break
    end
end
if length(candidate) == initial then
    // we did not find one to add
    solution = solution + candidate
    return solution // base case
end
remainder = remainder - selected
candidate = candidate + selected
// tlim is the timeout defined by the user
if $time > tlim$ then
    return solution
end
time = time + 1
if IsAdmissible(candidate) then
    solution = solution + TabuMove(candidate, remainder)
end
else
    candidate = candidate - selected
    tabu_list = tabu_list + selected
    remainder = remainder + selected
    solution = solution + TabuMove(candidate, remainder)
end
return solution
```

iteratively searches for improvements to the cardinality of an admissible option-set. It starts with a randomly chosen option, and adds (random) remaining options singly, checking for admissibility. If it reaches a point where no admissible options are found, it preserves the best result and randomly restarts. A *tabu list* prevents the searcher from re-tracing steps to the same point it just found. A *tabu tenure* details for how many moves that point will be considered *tabu.* One commonly lists the last few moves as *tabu.* If we conceive of Tabu search as a hill-climber, the tabu list acts to restrict which portion of the hill we can visit. An iteration limit ensures the algorithm terminates. IsAdmissible represents a call to the SAT solver with the given optionset merged with the existing admissible solution, R. Although it is not necessarily the case that there are single options which are admissible, in practice this is common.

If this is not the case, the algorithm performs a random search, beginning with 1-sets of options.

Evaluating the Improvements

I now present experimental results of the technique on the large goal model presented in §7.2.1. The ESP model contains 231 non-preferred nodes and 236 relations.

I demonstrate the utility and scalability of the technique by presenting evaluation results for this model in various configurations of options, using different pre-processing steps to reduce the problem scale (Table 7.1). The first row of results reflects the raw time it takes to evaluate this particular model with no options (using the SAT solver). The third column shows that adding the set cardinality heuristic (a call to Naive with a maximal set size of 8 and minimal size of 5), results in some improvement in running times. For example, with 15 options, the running time is approximately 30% shorter (while returning the same options, in this case). Similarly, while there is an exponential increase in evaluation time for the naive algorithm, Tabu_Search clearly follows a linear trend. However, the tradeoff is that Tabu_Search misses some of the solutions, although in the tests, assuming an average-case model configuration, it found half of the maximal sets of options.

Quality of solutions. While the naive approach returns all solutions (the Pareto-front of non-dominated solutions as shown in Fig. 7.2), the Tabu search heuristic will only return an approximation of this frontier. Tabu_Search returned, in the case with 15 options, one maximal set (of two), and 2 subsets. This in turn affected the number of alternatives that were found. Using the naive approach with 15 options, I identified 7 dominant solutions and 8 other alternatives (unrelated via preferences). Using Tabu_Search, with the smaller option sets, the algorithm only returns 4 dominant solutions and 6 other alternatives. These are individual solutions; I permit combinations, so the total number is much higher (the powerset). However, the greatly reduced running time will allow for more model exploration than the naive approach.

7.4.2 Evaluating the ATMS Approach

The ATMS approach to the requirements problem is again exponential for the MINIMAL-GOAL-ACHIEVEMENT operator. However, experiments on random requirements problems show that this limitation is not severe for typical early-phase requirements problems. I used random models to show this, but they are similar in size to industrial experiences, as explained in the beginning of §7.4. However, it is worth considering some optimizations.

Table 7.1: Comparing naïve vs. heuristic option selection. The fifth column represents the number of calls to the SAT solver; the last column indicates how many solutions the local search found: *max*, the number of maximal sets, *sub* the remaining non-optimal sets. I used a time step limit of 400.

Options	**Naive**	**Naive(8,5)**	Tabu-Search	# calls	Tabu solns
0	0.062 s	–	–	–	–
4	0.99s	-	0.08s	2800	all
6	3.97s	–	0.11	2800	1 max, 1 sub
9	31.8s	23.6s	0.14	2942	1 max, 2 sub
12	4m23s	3m6s	0.16	3050	1 max, 2 sub
15	33m	21m	0.18	3165	1 max, 2 sub
20	-	-	0.19	3430	1 max, 2 sub

Random Requirements Problems

I evaluated the ATMS implementation on large, randomly generated requirements models. I examine two empirical questions. The first concerns the size of model this approach can tackle.

The source code for my experiments is available at http://github.com/neilernst Techne-TMS. The experiments were run using (primarily) Clozure Lisp Version 1.5-r13651 on a Macbook Pro 2.4 Ghz, with 4 GiB of RAM. There are several opportunities for optimizing the code which I have not yet undertaken. I created random requirements models using a growing network attachment model Krapivsky and Redner [2001], where each new node is added to an existing node with a certain probability based on the number of existing attachments to that node. This produced a digraph where each node had out-degree of one. Since this seems unrealistic in a requirements model (where lower-level requirements might refine multiple higher-level requirements), I randomly added new edges between the nodes. Finally, I classified some of the edges in the tree as either alternatives (i.e., OR-refinements) or conflicts.

Table 7.2 shows the experimental results on random model permutations. Model sizes were chosen to represent a variety of model sizes according to industrial experiences with early requirements modeling. The **Load Time** column reflects the total time to calculate all minimal explanations in the model. Finding the results of MINIMAL-GOAL-ACHIEVEMENT for a given set of goals is polynomial, and therefore quite fast, so I have not shown those numbers.

There are three constraints on reasoning time. The first is obviously the overall size of the model, in terms of the number of nodes. The second is the number of assumptions, or tasks, which are used.

Nodes	**Tasks**	**Contradictions**	**Connectivity**	**Load Time**
50	30	10	5.00%	0.05s
100	66	20	2.70%	0.07s
100	83	4	2.60%	0.04s
150	99	6	1.70%	0.15s
150	100	30	1.70%	0.19s
200	135	8	1.20%	0.42s
200	131	40	1.20%	0.46s
250	162	10	0.95%	0.32s
250	166	50	0.95%	0.33s
300	200	12	0.79%	4.02s
300	195	60	0.79%	2.59s
400	265	80	0.59%	8.23s
500	335	20	0.46%	63.4s
600	398	12	0.33%	110.0s

Table 7.2: Label calculation times for requirements models using ATMS. Connectivity measures the mean percentage of the model to which a node is immediately connected.

The final constraint is the number of connections between nodes. Our results reflect this, with excellent performance for models with a few hundred nodes, and declining performance as more nodes are added. Contradictions serve to filter out explanations, and so act to restrict the exponentially large search space. The main constraint, however, is the number of assumption (task) nodes. As with the pQGM approach, the reasoning is exponential, as expected given the abductive nature of the algorithm. However, the time to find minimal explanations for all goals is feasible for a 600-node model.

Optimizing the ATMS

A worst-case model for the ATMS approach would be a model with no conflicts and no disjunctions. In that case, all possible explanations must be compared and rejected, and we cannot take advantage of the incremental nature of the ATMS. If the ATMS detects that the consequent for a refinement has the same label as the consequent, then the evaluation procedure "short-circuits" and finishes [de Kleer, 1986]. This will never happen when each consequent is represented by the union of its antecedent's explanations.

There are two ways that are promising for reducing running time. Clearly there is no obvious way to reduce the exponential algorithm. Instead, I propose to reduce the "apparent" problem size (i.e., N in the 2^N algorithm). The other approach is to concede optimality and use a heuristic technique to find local optima, as with Tabu search in §7.4.1.

Equisatisfiable Groups Consider the example in Fig. 7.7. Since all nodes in the implication chain, i.e., $\{A, B, A \wedge B \rightarrow C, C \rightarrow D\}$, are required (model 1), I replace the chain with two nodes: a single node that is the objective, and a meta-node representing the implication chain (model 2). This only works, of course, if the antecedents are not connected to other nodes (although they could conceivably participate in several chains). Conceptually, this is equivalent to converting a series of sub-requirements into a single abstraction. It is simple enough to "unwind" the meta-node to understand the chain of reasoning it captures.

Similarly, if all tasks are necessary for a given goal to be satisfied (case 2), they can safely be replaced with a single meta-task. Since our algorithm for abduction is (worst-case) exponential in the number of tasks (assumptions), this is a significant improvement (halving the worst case performance). It is also fairly common in requirements models; in our case study, I was able to reduce the payment card example from **159 tasks** to **62 tasks**. More importantly, this frees up memory, since the abductive explanation finding is exponential in space complexity for the worst-case. In my experiments, the biggest hurdle is availability of memory, since using virtual memory is so inefficient.

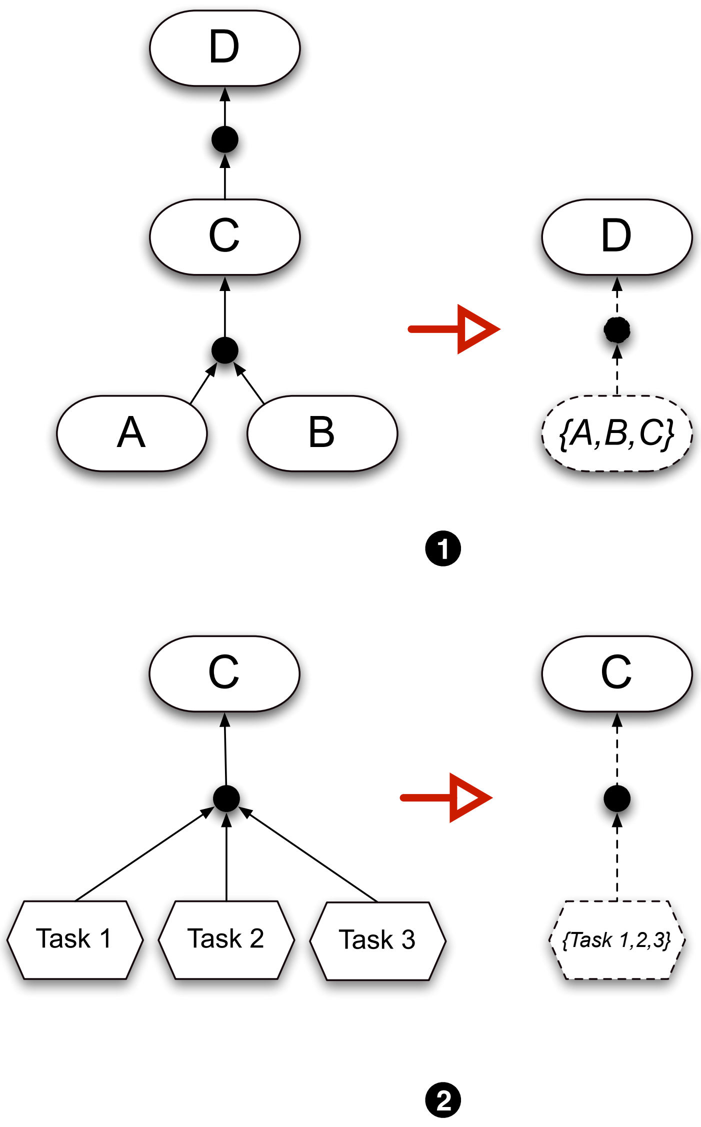


Figure 7.7: (1) Creating an Equisatisfiable Group; (2) Creating a Task Group.

This optimization parallels the DPLL algorithm's notion of unit propagation in propositional satisfiability [Davis et al., 1962]: if a set of clauses contains a single literal, then thst fact is used to either delete clauses containing the literal, or remove the literal's negation from clauses containing it.

Listing 2 shows the implementation in Common Lisp for equisatisfiability clustering. The entry point is the function (find-eqgrp-roots), which takes a graph representing an ATMS structure, and searches it for equisatisfiable groups. The predicate (eqgrp-p) takes a model and a vertex, and returns whether that graph forms an equisatisfiable group. An equisatisfiable group is defined as a graph with no contradiction nodes and no disjunctions.

```
(defun find-eqgrp-roots (graph)
  "account for our intermediate implication nodes. Does not account for alternatives.
   In other words, the model ((a^b)v(c^d)) -> e will return a,b,c,d"
  (let ((children (gather-non-link-nodes graph))) ;
    (loop for child in children
       if (and (eqgrp-p graph child)
               (not (taskp child)))
       collect child into subtrees
       finally (return subtrees))))

(defun eqgrp-p (graph vertex)
  "does the given vertex have or-children or contradictions?"
  (let ((kids (dfs graph vertex #'nothing))) ;; kids = list of child nodes
    (loop for kid in kids
       if (and (regular-node-p kid)
               (not (taskp kid))
               (> (target-edge-count kid) 1))
         ;; a non-meta node, that is not a task, with more than one incoming arc, is an OR node
         return nil
       finally (return t))))
```

Listing 2: LISP code for determining equisatisfiable sets

```
(defun collect-tasks (graph)
  "for the given graph, return sets of tasks which can be collapsed into one meta-task"
    (let ((leaves (find-leaves graph)))
    (let ((imps (find-leaf-implications leaves)))
      (loop for impl in imps
         collect (parent-vertexes impl) into tasks
         finally (return tasks)))))

(defun find-leaf-implications (leaves)
  "Find the implication nodes to which the leaves are connected"
  (loop for leaf in leaves
     append (child-vertexes leaf) into implications
     finally (return (make-set implications))))
```

Listing 3: LISP code for determining Task Groups

For task grouping, Listing 3, we must first find all tasks in the REKB; we then look for siblings of those leaves. Each sibling cluster can be replaced by a single temporary node to reduce the number of assumptions introduced (since all siblings must be true to satisfy the parent).

7.5 Other approaches

The purpose of the functional description of the requirements problem was to separate implementation and specification. Consequently, my use of the ATMS is not the only implementation to use for the REKB. For example, the CAKE reasoner that formed part of the Requirements Apprentice [Reubenstein and Waters, 1991] was based on the TMS of [Doyle, 1979]; while supporting incremental reasoning, CAKE does not handle minimal explanations, due to its use of TMS (as opposed to my use of ATMS, the abductive counterpart).

We have already seen how the SAT-based reasoners in Sebastiani et al. [2004] support the ARE-GOALS-ACHIEVED-FROM operation and, with appropriate selection of weights, the MINIMAL-GOAL-ACHIEVEMENT operation. In particular, my work in [Ernst et al., 2010] showed that a SAT engine can be combined with a problem solver to identify preferred solutions to the requirements problem. A different, approximate approach to solutions would to leverage the recent progress on SATisfiability testing, and try to find minimum-cost SAT assignments by repeated calls to SAT with random starts. Some SATisfiability solvers provide mechanisms for incremental reasoning, but stumble badly in the face of inconsistency, nor do they give explanations for a particular set of goals, e.g., MiniSAT [Een et al., 2010]. Another approach is to treat the SAT problem as a stochastic local search for a model [Selman et al., 1992], and use a greedy search technique like GSAT or WalkSAT [Selman et al., 1993] to quickly find a satisfying assignment (if one exists). A variant, MaxWalkSAT, can be used to solve weighted SAT problems like the MinSAT approach in Sebastiani et al.

Another weighted approach is to use pseudo-Boolean minimization [Boros and Hammer, 2002], which allows weights to be assigned to boolean formulas, and then looks for satisfying assignments that are maximal/minimal. The next step is to request minimal-cost assignments, corresponding to abduced sets, by weighing task atoms. Both this approach and the SAT approach require the REKB to be formulated as a completed propositional formula, in order to capture that the only way a goal can be satisfied is if it is entailed.

More interestingly, one can also define atoms representing formulas that indicate *changes of tasks becoming used* in a solution; giving these a weight of 1 would find solutions that minimize new tasks to be used in the solution of the new requirements. Unfortunately, the complexity of this problem may be even higher, and practical success depends on the precise form of the formulas encountered.

While I use abduction to find minimal sets of explanations for the satisfaction of the requirements problem, there have been a number of attempts to use abduction in requirements engineering to resolve inconsistencies. The work done in [Russo and Nuseibeh, 1999; Garcez et al., 2001; d'Avila Garcez et al., 2003] used abductive inference as diagnosis to resolve inconsistent requirements: if a requirements document is found to be inconsistent, abduction can identify the minimally inconsistent sets of requirements that have lead to that situation. This can then be used to repair the problem, and so evolve the requirements. Menzies et al. [1999] used abduction similarly, this time in order to combine multiple stakeholder models. Abductive reasoning is used to build consistent "worlds" based on the stakeholder views: a world that is most amenable to all views is the desired outcome.

Finally, the notion of solutions to requirements problems was the focus of Gay et al. [2009]. They looked at finding solutions to requirements models, and in particular, finding robust solutions. A robust solution is one that exists in a neighborhood of (nearly) equally preferred solutions. This means that in the event that something in the search changes, the initial solution is still a close approximation to the new optimal solution. The key to the success of this technique is the presence of numeric weights for a combination of goal, risk, and mitigation elements. This is combined to derive a single objective function that can be maximized. The notion of robustness is an interesting one to consider in selecting solutions to requirements problems.

Chapter 8

Solving Evolving Requirements Problems

Up to this point, I have focused on consistent and static Requirements Problems. Chapter 6 defined the specification-level definition of the Requirements Evolution Problem, and this chapter considers implementation-level strategies for managing Requirements Evolution Problems. I only examine those changes which are unanticipated. By 'unanticipated' I mean that there exists no mechanism in the implementation specification S to accommodate these changes (cf. §1.1). This clearly positions the Requirements Evolution Problem as distinct from the Self Adaptation Problem. Let me recapitulate the formal statement of the Requirements Evolution Problem (REP):

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

Note that $\hat{S}$ is no longer required to be minimal.

The Self Adaptation Problem (SAP) is quite different. With respect to the aspects of G, D, 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. Now, there is a caveat to this, which is that with a suitably flexible framework, an adaptive specification can be used to select services that satisfy our changed goals or domain assumptions. Since these services can be quite heterogeneous, then in a sense there is a continuum between adapting and evolving. The essential distinction is the extent to which the changes are anticipated.

Cheng et al. [2005] have a similar perspective. They divided RE into four levels according to what undertakes the RE (a human or a machine) and how the RE was done. Level 1 is humans anticipating possible events in domain D to design a set of implementations S. At level 2, the implementation S is 'self-aware', and selects possible adaptation implementations S_i in S to manage changes in the domain. The transition from the SAP (levels 1 and 2) into the REP happens at level 3, which occurs when S "may be presented totally unanticipated input D, such that SAR's Level 2 RE fails to adapt." (Level 4 is the problem-theoretic investigation into adaptation mechanisms).

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**.

In this chapter, as with the previous chapter, I assume that we work with a consistent REKB. I begin by describing a sample methodology for managing changing requirements problems. I then focus on the problem of revising the REKB, and describe the GET-MIN-CHANGE-TASK-ENTAILING operation that supports this. A particularly important question is the set of properties we can use to select new solutions. To manage changes, we need tool support for version control [Ernst et al., 2008]. I evaluate the feasibility of the GET-MIN-CHANGE-TASK-ENTAILING operator along with the version control operations, and conclude by examining other approaches.

8.1 Methodological Guidance for Solving Unanticipated Changes

Since the focus of the REP is changing systems, it behooves me to outline the process by which these changes occur, as well as the impact the changes have on the requirements problem. Fig. 8.1 outlines these steps in graphical form.

Step 1a. Elicit requirements from stakeholders and map the speech acts using the CORE ontology 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

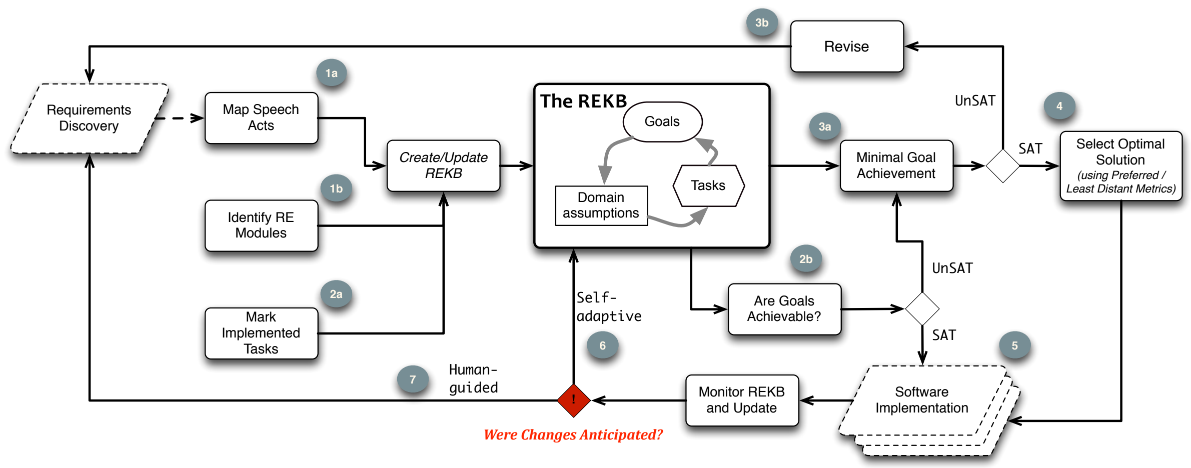


Figure 8.1: A methodology for Requirements Evolution Problems

Verifone) for the 1,200 outlet grocery store chain, you will want to add the details of the networks (perhaps Verifone is responsible for all payment-processing).

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 using GET_ALL_PREFERRED. 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 as defined in [Wang et al., 2009].

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 Cheng et al. [2005], 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.

I now turn to the issues involved in revising our REKB (step 7).

8.2 Revising requirements

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). What do I mean by a "revised REKB"? At its most basic, it is exactly what is described in the (semi) formal description of the REKB: $\delta(G), \delta(D)$ and $\delta(S)$. Recall that an REKB is a tuple $\langle \text{REKB.TH}, \text{REKB.ST} \rangle$, the set of asserted theories and introduced (but not asserted) atoms in the symbol table. A well-formed formula (wff) in REKB.TH is an atom associated with a sort, which classifies each atom as one of goal, domain assumption, or task. Therefore revision means that any of these formulas can change.

The key issue is what the outcome of this change is for each component. Revisions to the symbol table do not affect the state of the requirements problem until they are introduced with an assertion. Revising a wff in REKB.TH implicitly updates the symbol table. Revision of the REKB can be accomplished using the assert, declare and retract operations. If we try to declare or assert an atom with an existing label in the symbol table, the REKB silently updates the state of the REKB using the Levi identity [Levi, 1977]: revising the REKB with ϕ is equivalent to retracting $\neg\phi$, ensuring the REKB is consistent, and then expanding by ϕ.

The concept of belief revision in artificial intelligence (cf. §2.2.6) closely parallels the concept of REKB revision. And yet, there are important distinctions that mean that we do not accept the belief revision literature in its entirety.

8.2.1 Requirements Revision

There are three key principles in (AGM-style) 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[1]. 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

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

"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.

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.

Consider the revisions used in Zowghi and Offen [1997], which is classical belief revision combined with Poole's default logic (cf. §9.6.1). The logic is not focused on the requirements problem. Revision is theory manipulation, and the only connection to the requirements model is two-fold: the selection of the epistemic entrenchment order, and the ability to 'downgrade' requirements to default (preferred) status rather than 'mandatory' status. It is possible to re-order the entrenchment to prioritize certain goals, but this requires us to consider all elements of the problem (that is, the tasks, goals and assumptions) equally. Zowghi and Offen do things opposite to what I am claiming: when presented with a new piece of information, the requirements are relaxed if that is necessary to accommodate the revision. My approach is to examine the requirements and accept the new information *only* if it leads to a better solution.

My approach to the problem of revising the REKB is to view it from the perspective of solving the requirements problem. The key question after learning of a potential change to the REKB is to use GET-MIN-CHANGE-TASK-ENTAILING (Step 4 of the methodology) in order to find new solutions S', even if the new set of goals have changed. This is particularly important in the context of iterative software development, since in this methodology, customers accept that part of software development is to acquire more knowledge about the possible solutions (and presumably will not be enraged when a goal is rejected!).

8.3 Finding Minimal New Changes

Recall the definition of the GET-MIN-CHANGE-TASK-ENTAILING operator from Chapter 6: it 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 solutions to the requirements problem with respect to a distance function. Alg. 9 shows how we calculate the minimum change in the implementation.

Algorithm 9: GET-MINIMAL-CHANGE-TASK-ENTAILING

```
Input: goalsG :$\wp$(GOALS) × originalSoln :$\wp$(TASKS) × DIST − FN
Output: newSolns :$\wp$($\wp$(TASKS))
possSolns = MINIMAL-GOAL-ACHIEVEMENT(goalsG)
// Find the sets of possible solutions using MINIMAL-GOAL-ACHIEVEMENT
foreach $s \in$ possSoln do
    newSolns = DIST_FUN(possSoln,originalSoln)
end
return newSolns
```

The important consideration in choosing new solutions is the choice of distance function, so let us examine some possible choices.

8.3.1 Properties for Selecting New Solutions

Having defined how our language will handle inconsistency, we can now turn to the question of finding answers in changing models. In particular, a central task when updating the system is to decide which of the new solutions is best, and in particular, is best with respect to what has gone before. It seems intuitive that we do not want to completely ignore previous implementations. Requirements re-use is generally seen as a good idea [Ferreira et al., 2010]; there is also the “Not-Invented-Here” syndrome, widely seen as an anti-pattern.

So what criteria are important in selecting this new solution? I defined several properties Π in [Ernst et al., 2011], 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 $\mid \hat{S} - S_0 \mid$ (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 $\mid \hat{S} \mid$ (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. The details of these functions is captured in the source listing of Listing 4. I pass these functions in to GET-MIN-CHANGE-TASK-ENTAILING in order to compare the sets. Note that in this implementation a single set that satisfies the comparison is returned, whereas in practice many such sets might conceivably exist. The subsequent analysis would be to compare these sets for preference optimality, as I did in §7.2.1.

```
(defun min-effort (new-solutions)
  " The solution which differs in the fewest tasks"
  (let ((existing (getf (first *impl-repo*) :TASKS )) (smallest (first new-solutions)))
    ;;assume use last set of impl
    (loop for soln in new-solutions
       if (< (length (set-difference soln existing)) (length (set-difference smallest existing)))
         do (setf smallest soln)
       finally (return smallest))))

(defun max-fam (new-solutions)
  "The solution which differs in the fewest tasks from the previous one"
  ;; maximize intersection
  (let ((existing (getf (first *impl-repo*) :TASKS )) (biggest (first new-solutions)))
    ;;assume use last set of impl
    (loop for soln in new-solutions
       if (> (length (intersection existing soln)) (length (intersection existing biggest)))
         do (setf biggest soln)
       finally (return biggest))))

(defun simple (new-solutions)
  "The shortest new solution, if a match, determined non-deterministically"
  (let ((sorted (sort new-solutions #'shorterp)))
    (first sorted)))
```

Listing 4: LISP code for calculating distance using the three properties.

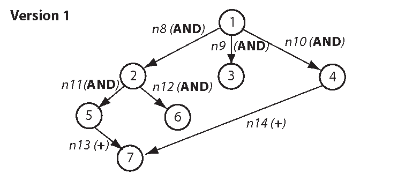
8.4 Version Control of Requirements Problems

In order to reason about what has changed in the model, we need mechanisms to support basic merging, matching and version control. I now look at how we can support those abstractions using some custom tool-support my colleagues and I described in Ernst et al. [2008]. This was a collaborative project in which the fourth author, Nguyen, was responsible for the Molhado infrastructure, the third author Yu wrote code to do the mirroring, and I created the case studies, built the user interface, created the repository, and wrote the paper.

Instead of building a version control system on top of a text-oriented system such as CVS, I have used a customizable, object-oriented, configuration management (CM) infrastructure, Molhado [Nguyen et al., 2005]. In file-oriented approaches, developers have to map the versions of high-level logical entities (e.g., classes, goals) into versions of files in a file system. This causes cognitive dissonance as developers translate between the problem domain and the development environment. Object-oriented CM allows developers to remain focused on the problem domain.

Molhado has been previously described in the literature [Nguyen et al., 2005; Dig et al., 2007]. I

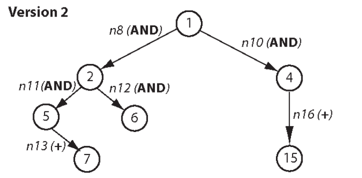
Version 1



Attribute table

node	"type"	"source"	"sink"	"children"	"name"	"entity"
n1	*node*	*undef*	*undef*	[n8,n9,n10]	"Ticket"	Hard-goal
n2	*node*	*undef*	*undef*	[n11,n12]	"ImproveEmail.."	Hard-goal
n3	*node*	*undef*	*undef*	null	"AssignRespon.."	Hard-goal
n4	*node*	*undef*	*undef*	[n14]	"QueryInterfa.."	Hard-goal
n5	*node*	*undef*	*undef*	[n13]	"UseSMTPAuth"	Hard-goal
n6	*node*	*undef*	*undef*	null	'UsernameAre.."	Hard-goal
n7	*node*	*undef*	*undef*	null	"Security"	Soft-goal
n8	*edge*	n1	n2	*undef*	"AND"	And-rel
n9	*edge*	n1	n3	*undef*	"AND"	And-rel
n10	*edge*	n1	n4	*undef*	"AND"	And-rel
n11	*edge*	n2	n5	*undef*	"AND"	And-rel
n12	*edge*	n2	n6	*undef*	"AND"	And-rel
n13	*edge*	n5	n7	*undef*	"+"	Plus-rel
n14	*edge*	n4	n7	*undef*	"+"	Plus-rel

Version 2



Attribute table

node	"type"	"source"	"sink"	"children"	"name"	....
n1	*node*	*undef*	*undef*	[n8,n10]	"Ticket"	
n2	*node*	*undef*	*undef*	[n11,n12]	"ImproveEmail.."	
n3	*undef*	*undef*	*undef*	*undef*	*undef*	
n4	*node*	*undef*	*undef*	[n16]	"QueryInterfa.."	
n5	*node*	*undef*	*undef*	[n13]	"UseSMTPAuth"	
n6	*node*	*undef*	*undef*	null	'UsernameAre.."	
n7	*node*	*undef*	*undef*	null	"Security"	
n8	*edge*	n1	n2	*undef*	"AND"	
n9	*undef*	*undef*	*undef*	*undef*	*undef*	
n10	*edge*	n1	n4	*undef*	"AND"	
n11	*edge*	n2	n5	*undef*	"AND"	
n12	*edge*	n2	n6	*undef*	"AND"	
n13	*edge*	n5	n7	*undef*	"+"	
n14	*undef*	*undef*	*undef*	*undef*	*undef*	
n15	*node*	*undef*	*undef*	null	"Usability"	
n16	*edge*	n4	n15	*undef*	"+"	

Figure 8.2: Graph-based version control

will concentrate on the extensions we added to support model-based versioning in the context of a requirements goal model. The Molhado infrastructure was customized with a model-driven, graph-based change management approach. This was then combined with a goal-modeling tool created using Eclipse model-driven development frameworks [Gronback, 2009].

On top of the Molhado CM infrastructure, an extended version of the OpenOME tool did the actual modelling.

8.4.1 Mirroring

Let us now examine the data structures used for requirements versioning. A goal model is defined as $G = \langle id, g, r \rangle$. To represent the properties associated with a goal model entity, I use Molhado's slot-based property mechanism. A node has a set of *slots* that are attached to it by means of *attributes*. In other words, slots hold attribute values and attributes map nodes to slots. Nodes, slots and attributes that are related to each other form attribute tables, whose rows correspond to nodes and columns to attributes (see bottom parts of Fig. 8.2). The cells of attribute tables are slots. Version control is added into the data model by a third dimension in the attribute tables. Slots in any data type can be versioned using such data structures. To support model-driven CM for goal models, I represent their internal structure with a directed graph (possibly multi-graphs, but not hyper-graphs).

Figure 8.2 displays two versions of a goal model. In the new version, the attribute table was updated to reflect the changes to the graph structure as well as to the slot values. For example, since node 3 and edges corresponding to "n9" and "n14" were removed, any request to attribute values associated with those nodes will result in an undefined value. On the other hand, node 15 and the edge corresponding to “n16” were inserted, thus, one new “node” node (“n15”) and a new “edge” node (“n16”) were added into the table. Attribute values of these nodes were updated to reflect new connections. Attribute values of existing nodes were also modified. For example, the “children” slot of “n1” now contains only “n8” and “n10”, since the edge “n9” were removed.

8.4.2 Versioning

Changes to a goal model’s entities in OpenOME are mirrored and updated to the graph structures in the model-driven CM repository. The mirror maintains a mapping between the model in memory and the model in persistent storage. The mirror mapping is necessary as the EMF-generated model does not use the Molhado-specific in-memory data structure. In OpenOME, such a mapping is implemented as follows.

For each goal modeling project, the mirror contains (with decreasing granularity): 1) a folder object, representing the project name; 2) leaf folders containing model objects that are uniquely identified by name of goal model files; 3) model objects contain Molhado graph structure objects (i.e., nodes and edges) that maintain a one-to-one mapping with the goal model objects in the EMF model. In other words, not only the versions of files but also the versions of individual objects are being maintained.

The types and names of intentional elements are stored in Molhado as string attribute slots of the nodes. The children of decompositions or targets of contributions (the goal-model specific concepts) are stored as object attributes in the Molhado node objects. The types of decompositions or contributions are stored in Molhado as a string attribute slot. Finally, the parent/child of decompositions or source/target of contributions are stored as object attributes in the Molhado edge objects.

Five algorithms implement the version control functionality.

Algorithm 10 – Converts an EMF model into mirrored objects that are ready to be checked into the Molhado repository. This algorithm, shown in Alg. 10, implements the INITIALIZE-REPOSITORY and TEMPORAL-COMMIT operations. The procedure iterates through the EMF object twice. The first pass picks out the node elements to form a mapping between the intentional elements (e.g., goals) and their mirroring nodes. The second pass mirrors edges (e.g., goal decompositions) using the mapping between nodes and their mirrors. Note that when constructing a mirror for each edge, the source/target nodes should have been constructed. Therefore, the two passes cannot be fused due to the fact that the iteration in EMF models are not ordered by type of elements.

Algorithm 11 – Alg. 11 checks out a version of the model from the Molhado repository and converts it into an EMF goal model in memory. This is an inverse **commit** operation, also called CHECKOUT.

Algorithm 10: TEMPORAL-COMMIT

```
Input: $e$, an EMF goal model in OpenOME
Output: $m$, objects mirroring $e$ in Molhado and $M$, a mapping between objects and their mirror
$M = \{\}$
foreach node $o \in e$ do
    create $m_o$ to record the type and name of $o$
    $M = M \cup \{o, m_o\}$
end
foreach link $o \in e$ do
    $p$ = source($o$), $c$ = target($o$)
    $m_p = M(p)$, $m_c = M(c)$
    create an edge $m_e$ to connect $m_p$ with $m_c$
    add $m_e$ as a child to $m_p$
end
return $M, m$
```

Algorithm 11: CHECKOUT

```
Input: $m$: a mirror of a versioned model in Molhado
Output: $e$, an EMF goal model in OpenOME and $M'$, mapping from mirrored object to objects
$M' = \{\}$
$e = \{\}$
foreach node $m_o \in m$ do
    $o$ = Factory.createGoal($m_o$.name)
    $M' = M' \cup \{< m_o, o >\}$
    $e = e \cup \{o\}$
end
foreach edge $m_o$ in $m$ do
    $m_s$ = source($m_o$), $m_t$ = target($m_o$)
    $o$ = Factory.createLink($m_o$.type)
    source($o$) = $m_s$, target($o$)= $m_t$
    $e = e \cup \{o\}$
end
return $M', e$
```

Algorithm 12: CREATE-MODEL-IN-REPO

```
Input: $f$: an EMF file, $R$: the repository of project
Output: $R'$, updated repository by version $v$ of $f$ where current_version$(R, f) = v$
$n = f.name$
$e =$ CREATE-MODEL$(f)$
$v =$ FETCH-VERSION$(R, f, n)$
if $v > 0$ then m was previously committed to R
    $m, M' =$ Checkout$(R, n, v)$
    $M = \{(o, m_o) \mid (m_o, o) \in M'\}$ // invert the mapping
    $v = v + 1$
    $R' = R$
end
else
    $m, M =$ CREATE-MIRROR-FROM-EMF$(e)$ // Algorithm 10
    $v = 1$
    $R' = checkin(R, m)$
end
return $R'$
```

There are two issues to keep in mind: an existence dependency between nodes and edges (an edge must connect two existing nodes); and unordered elements in the mirrored objects. To accommodate these, as in Algorithm 10 we must construct the mapping between node objects first, then update the relations on the edge objects.

Algorithm 12 – Alg. 12 explains how a model is committed to the Molhado repository whenever a goal model is opened in the editor. This procedure modifies the `CreateModel` method in the EMF-generated class `GoalmodelEditor`. The algorithm first checks the version of a model identified by the file name. If the version number is positive, then the model is already stored in the repository. In this case, the mirrored data structure in Molhado is checked out with an increment to the version number. Otherwise, if version number is zero, then it has never been checked in. In that case, we convert the parsed EMF model into the data structure for Molhado repository using Algorithm 10.

8.4.3 Diff

The diff operation compares two models and generates a set of operations that can be used to generate one model from the other.

Algorithm 12 – Modifies the `doSave` operation in the generated EMF editor such that the edited model is checked in the repository after being saved. A naïve solution would be to recreate a mirror of the edited EMF model using Algorithm 1, then *checkin* the mirror into the repository. However, such a *commit* would lose track of the version information for individual objects. I want to avoid situations where a goal rename is treated as a pair of delete/create operations, since this obscures the semantics

of the renaming operation. To manage this scenario, I do not treat an editing session (i.e., the edits that occur in between 'save' operations) atomically, that is, only committing the final result of the edits. Instead, I reuse the editing command stack generated in the EMF goal model editor (an Eclipse feature) in order to replay (redo) the editing changes while updating the mirrored objects accordingly. Essentially, our algorithm saves all changes the user makes during the session. After all the editing commands are performed on the mirror, we check in the modified mirror objects such that changes to individual elements are traced in the repository.

Algorithm 13 – When different versions of the goal graphs reflect changes originating from other sources (code, documentation) through reverse engineering, I cannot guarantee that the editing transactions are completely recorded. For this scenario, one would like to present the user with a report on what differences exist. To support such cases this algorithm computes the difference of two goal models, one from an existing version in the repository E_1, the other from the one currently loaded inside the graph editor E_2. As the Molhado repository fully supports object-oriented versioning, we need only uniquely identify each object to ensure all changes are saved.

Consider the case when a goal g_1 in the EMF model E_1 is refined into g_2 and g_3 in E_2. As long as I identify that the parents of g_2 and g_3 have the same name as g_1, the id of g_1 in the mirror of E_1 will be the same as that in E_2. Even when g_2 and g_3 change their ordering in the serialized model in E_3, another version of the model, Algorithm 5 will still treat them as the same model, without notifying Molhado to increment the version identifier. Thus, only semantically relevant changes will be version controlled. This is an important advantage over traditional text-based CM such as CVS, where any differences in the serialized models, such as a difference in the ordering of subgoals between g_2 and g_3, would be considered a change.

A limitation of the diff-based versioning algorithms (including Algorithm 13) is that one can not trace the renaming of the *id* of an object. For example, if g_1 in E_1 is renamed to g_1' in E_2', then g_1' is considered a new goal by Algorithm 13, and its subgoals need to first be disconnected from g_1, then reconnected to the new goal, g_1'. Without recording the editing operations, one has to rely on techniques such as Origin Analysis [Godfrey and Zou, 2005] to identify such changes on basis of the context of neighbouring nodes. Since the context can be changed as well, such an approach is imprecise. However, the editor-based version control (Algorithm 12) can precisely catch such changes by simply renaming the object. Model versioning is not the primary focus of the work. However, the approach in Algorithm 12 is a novel way to preserve the context of edit histories.

8.4.4 Reporting

Another component of the tool chain is change analysis of the goal model versions. One of the basic features in a CM system is a reporting tool which displays changes between different versions of an artifact. Since the CM is model-driven, a reporting and differencing scheme that addresses domain elements, rather than typical line-oriented differencing algorithms can be used.

The reporting tool first must detect changes. Suppose that we have two versions V_1 and V_2. The question is how to determine if a node or an edge in a graph has been deleted, inserted, or moved, and if an attribute table associated with it has been modified. To detect if an attribute table and its values have been changed, one attaches a "dirty" bit to a slot (containing an attribute value). Note that attribute-value functions would be called to modify attribute values in an attribute table. The bit will be set to "true" to signify a change. To detect the deletion of a node or an edge from a directed graph, the values of structural attributes (i.e. "source", "sink", and "children") are examined. A special value (undefined) signifies a deletion. The removed node (or edge) is not permanently deleted in the repository since it still exists in previous versions. The insertion of a node or an edge to a directed graph is signified by the appearance of a new row in the attribute table. To detect the relocation of a sub-graph, examine the change of the "source" value of an edge. If the "source" slot refers to a different node, the sub-graph starting from the "sink" node of that edge is relocated. These detection functions are applicable to any node and edge. A function to return differences between two arbitrary versions of an attributed graph was also implemented.

There are several characteristics of the framework that make structure-oriented differencing scheme simple, efficient, and accurate. The first one is the unique identifiers of nodes and edges in directed graphs. Second, the unique identifiers for nodes/edges are immutable. Third, the actual development history is accessible since the library functions for graphs will update values of slots whenever design objects or data records in the application are modified in the editors. Finally, using structured editors will preserve the identifiers of nodes and edges. Changes that were actually performed from one version to another can be easily reconstructed and reported by pair-wise comparisons of versions without dealing with sequences of actual operations explicitly.

This scheme is also efficient because it does not use complex directed graph comparison algorithms as in many existing differencing tools for design diagrams. No file content analysis is required. The reason is that the system works on the attributed, directed graph representation, rather than on the textual representation or a database representation of a diagram. Each element in the diagram has a unique identifier, which facilitates the detection of structural changes and subsequent reporting. Every change to the representation graph of the diagram is recorded as described earlier. Structural changes are stored as "parent", "children", "source", and "sink" slots. Generally speaking, the user interface module of the system only needs to retrieve structural changes and displays them. No add-ons to the CM system are needed to derive structural changes. Reporting changes is a matter of traversing one single graph and identifying changed elements.

8.5 Feasibility

8.5.1 Evaluating Requirements Problem Versioning

The implementation claims to manage goal models by versioning them and providing queries of the changes and change types between versions. To show that it is a feasible requirements model management tool, I provide a preliminary validation of the tool using a proof-of-concept case study. I focus on three dimensions in the validation:

- Scalability – Will the tool be able to handle meaningfully sized requirements models?
- Minimality – How much information is presented to the user?
- Conformance – Does the implementation preserve all information upon serialization?

Minimality measures the number of changed artifacts between versions, represented as a changeset. Minimizing changeset size makes it easier to track 'what' has changed. *Conformance* reflects how closely the changes represented capture what actually occurred. The more conforming a changeset is, the easier it is to track 'how' the model has changed.

Methodology – For the case study, I looked at a web-based tool for the management of software projects. One of the most well-known such applications is Sourceforge. I studied three versions of Trac, an open-source, Python-based project management tool[2]. Trac provides integration with Subversion, wiki-based web pages and ticket/bug reports, among other features. The first configuration I looked at is version 0.9.0, revision 2438, released October 31, 2005. The second example is Trac 0.10.0, revision 3803, released September 28, 2006. The final example is the latest development version of the Trac application, version 0.11dev, revision 5999, released September 7, 2007. There are 332 days between 0.9 and 0.10, and 344 days from 0.10 to 0.11dev. To conduct the evaluation, I extracted a series of goal models from the Trac source, generated using a technique similar to Yu et al. [2005b]. Characteristics of these models are shown in Table 8.1.

[2]trac.edgewall.org

version	# nodes	# edges
Trac09-low	1100	1193
Trac10-low	1309	1426
Trac11-low	1850	2015

Table 8.1: Basic statistics for the extracted goal models

Scalability – I assess the space and time complexity of the approach, to evaluate scalability. The tool is scalable so long as any one operation is less than $\mathcal{O}(n^2)$ in complexity.

Algorithms 9, 10, and 11 are linear in the size of the goal model. Both Algorithm 9 and 10 are implemented on top of existing routines in the Eclipse-generated tooling. Algorithm 9 adds overhead to a "doSave" operation in the EMF editor, and Algorithm 10 adds overhead to a "doSaveDocument" operation in the GMF editor.

Algorithm 12 has time complexity of $T_{bookkeep} + T_{redo}$ in addition to T_{save}.

$$T_{bookkeep} = \mathcal{O}(|s| \cdot (1)) \cdot T_{compare}$$

where s is the command stack, $|s|$ is the size of the stack, which may be larger than $|e|$, the size (numbers of nodes plus edges) of EMF goal model e; and $T_{compare}$ is the constant time operation to compare two identifiers.

$$T_{redo} = \mathcal{O}(|s|T_{op})$$

The time to redo all editing operations kept on the stack is a product of the size of that stack and T_{op}, the average time for one basic editing operation.

To elaborate: 1) the loop `foreach c in s` will iterate over all editing operations in the command stack s, and redo them while keeping track of the changes; 2) the `switch` statement will invoke one to two mapping lookups $M(o)$, depending on the type of operation (e.g., rename, delete, add). A lookup checks a hashtable that maps from an EMF object to a mirrored object, which takes $\mathcal{O}(1)$ comparisons.

Algorithm 13 has complexity of

$$\mathcal{O}(\log_2 |e| + (1) \cdot |e'|) \cdot T_{compare}$$

where $|e'|$ is the number of changed elements in the goal model e', and $|e|$ the total number of elements. No redo operations are required, since we do not implement the editing stack comparison in this algorithm. The algorithm finds the changed elements in the goal model in $\log_2 |e|$ time, then for each $|e'|$ changed elements, update the mapping in constant time.

comparison	(+)nodes	(-)nodes	(+)edges	(-)edges
$V_1 \rightarrow V_2$	404	195	467	234
$V_2 \rightarrow V_3$	957	416	1087	498
$V_1 \rightarrow V_3$	1222	472	1396	574

Table 8.2: Change statistics for the three low-level models using the tool. V_1 = Trac09, V_2 = Trac10, V_3 = Trac 11.

The space complexity for Algorithm 12 is in the worst-case two mirrors m, m', two EMF models e, e' and one mapping table M, plus an editing stack s. In other words, $5|e| + |s|$, where $|e|$ is the size of the goal graph e and $|s|$ is the size of the stack s. Since there is no need to store the editing commands s, the space complexity for Algorithm 13 is $5|e|$. For large models this could present problematic demands for memory, particularly in Eclipse, so persistent storage alternatives will be important.

Minimality – To measure minimality, I capture change statistics, *i.e.*, elements added, deleted, or changed. What an element is varies; in our tool, elements are atomic goals, decompositions, and contributions. In diff, elements are lines. In EMF Compare[3], a model-comparison plugin, the elements are EMF model elements, essentially equivalent to our goals and relationships. If a tool presents the users with too many artifacts in the reporting on operations, there may be other alternatives that are more useful.

I evaluate this solution relative to the standard line comparison tool, diff, an XML-based diff tool, Altova DiffDog[4], and EMF Compare. Note that file-based tools I compare must be used in concert with existing CM tools such as Subversion, while my approach is stand-alone.

Using the built-in reporting functionality, I produced raw data on the number of elements changed, added or deleted. Table 8.2 presents the results of comparing changes between the extracted structural versions of these models.

Let us compare results to `diff`. I ran the diff utility (ignoring whitespace) on the three versions of the goalmodel text representations (which are stored in XML format), and counted the number of reported changes. There was a negligible cost to perform the diff.

The result of running `diff`, shown in Table 8.3, is a much greater number of changes than our approach. On average, each node is represented by 3.94 lines in the XMI format. However, the resulting diff has a ratio of changed lines to changed domain elements greater than 8.35, double the ratio of average lines

[3]http://www.eclipse.org/emf/compare/
[4]http://www.altova.com/diffdog/diff-merge-tool.html

comparison	(+)lines	(-)lines
$V_1 \rightarrow V_2$	3375	2309
$V_2 \rightarrow V_3$	6531	3776
$V_1 \rightarrow V_3$	7030	3209

Table 8.3: `Diff` results for low-level goal models. Version numbers are as in Table 8.2.

for each node. This implies that for every changed element in the model, there are eight changes in the text file. Furthermore, the nature of the report diff issues is entirely syntactic, while my result is a set of changes described in the semantics of the domain model. Using an XML-aware diff tool produced results that were difficult to interpret, since elements that were deleted were often treated as renames. I conclude that text and XML diffs are not useful approaches.

From a model-based diff perspective, I also tested the Eclipse EMF Compare tool, a model-based comparison framework. I compared the Trac09.goalmodel file (an EMF-based XMI file) with the Trac10.goalmodel file. Each file contains over 2000 elements. I ended the compare step prematurely, as after 80 minutes of processing time the progress indicator was approximately 20% finished. By comparison, my largest example requires just over 4 minutes to produce. Using EMF Compare on models of this size is currently infeasible.

These results show that the proposed solution does not overwhelm the users with extraneous tool-specific detail.

Conformance – To show that the implementation maintains change history conformance, I need to demonstrate that the tool can 'replay' a series of changes in a diff D (see the definition of the diff operation in §8.4.3). For the scenario where there are two models and then one uses a tool to generate the diff, whether D represents the actual edit steps is impossible to verify, since there is no intermediate representation (and there are infinite possible paths). I verified this trivially by computing the difference for two related yet different models G_1, G_2, capturing the diff between them, and 'replaying' the operations in the diff to recreate G_2. This also confirmed that the algorithms do not alter the models beyond the operations the user specified.

I also evaluated Algorithm 12 (see 8.4.3) on its ability to improve on traditional diff techniques by 'replaying' full edit histories (where the previous evaluation was agnostic as to intermediate changes). I illustrate the viability of the approach with a short example, and then compare this to the results returned from EMF Compare. Figure 8.3 shows the changes. I start with a single goalmodel, V1: 3 goals, A, B, C, and two AND-decompositions relating them; then link A to a goal D, to produce V2;

finally, delete node B is to redirect one of the AND-decompositions. The tool reports the changes from $V1 \rightarrow V2$ as a new node and link, and a change in A; modifications in $V2 \rightarrow V3$ as a delete and two decomposition link changes (change to the source and the target of the relation).

When I compare V1 and V3 using EMF Compare, as one might if one did not store the intermediate V2 in, say, CVS, the tool detects 5 changes: the file name, the name of element D (from B), and the reference from A to B (now C to A; 3 changes). From a model comparison perspective, this is useful behaviour. However, I argue the information lost in not tracking the interim changes of V2 results in a significant loss of context about the requirements development process.

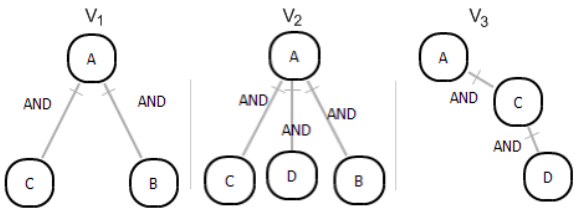


Figure 8.3: Forward-engineering validation

8.5.2 Evaluating Incremental Reasoning

As we saw in §7.4.2, the ATMS reasoner takes a long time to initialize the model, because it is calculating all minimal explanations for each node. This is a deliberate trade-off with the benefit of being able to quickly calculate incremental changes to the model. Adding new information to a requirements problem is handled quickly and generating new solutions for RP2 and comparing them to the old solutions from RP1 is likewise fast. My tool therefore supports some degree of model exploration and scenario analysis. In the previous evaluation I focused on the scalability of the naive approach. Now I will look at the benefits to using an incremental algorithm for updates, to support requirements model revision.

- **high-level**: new mandatory goals and refinements are added. This example has 4 new mandatory goals with 8 refinements;
- **new-task**: new tasks are available. Consists of 10 new operationalizations;
- **conflict**: stakeholders identify more contradictions. This example contains 15 new contradictions.

I evaluated the performance of two operations with these scenarios. The first operation measures how long it takes to load the new model, either starting again or incrementally; the second concerns how long

Scenario	Naive add (s)	Incremental add (s)	Min_Goal_Achieve (s)
high-level	1.89	0.070	0.029
new-task	2.49	0.620	0.130
conflict	1.91	0.048	0.023

Table 8.4: Incremental operations on a large requirements model (n=400). **Naive add** is the time taken to evaluate the scenario plus the original model; **incremental add** uses incremental support in the ATMS; **Min_Goal_Achieve** is the time it takes to identify the minimal tasks to satisfy mandatory goals.

it takes to answer MINIMAL_GOAL_ACHIEVEMENT for some set of mandatory goals. Here, stakeholders might be asking whether the new updates can still produce a viable solution.

Comparisons of the distance function are omitted since they are fast, linear time operations.

Table 8.4 shows the results. The numbers suggest that the incremental algorithm constitutes a clear improvement on starting from scratch. For example, looking for alternative solutions can be done nearly instantly, allowing stakeholders to use the REKB tool as a workbench for solution identification. While the naive algorithm (adding new changes to the REKB and re-calculating the labels) is not terribly slow, there is a large relative difference I expect to see in larger models as well.

The timing results for finding minimal new changes are also nearly-instant, allowing the REKB to support interactive decision-making.

Along the lines of the optimizations introduced in §7.4.2 one could imagine using the property of a requirements change-proneness to modularize the recalculations required. If we knew (guessed) a priori which requirements were likely to change, we could isolate those in a separate module. I have not explored this in more detail, however.

8.6 Related Work in Evolving Requirements

There are alternative approaches to evolving requirements problems. One approach is obviously completely manual: requirements re-use is the (typically unsuccessful) attempt to re-use the earlier requirements in a new or extended version of the initial design. If we favour a more automated approach, then early work on this problem has its roots in the requirements monitoring work of Fickas and Feather [1995]. Using goal models to drive adaptation was the focus of Liaskos et al. [2006] and more recently Wang and Mylopoulos [2009]. Goal models, since they reflect stakeholder desires, are used to guide variability decisions in the implementation. Liaskos et al. used this variability to support the changing (yet anticipated) needs of geriatric patients, for example, increasing font sizes on the application as eyes grew older.

One class of approaches deals with the Self-Adaptation Problem. The requirements-specific version of this problem is also known as requirements at run-time or requirements reflection [Sawyer et al., 2010], and the key objective is to bring requirements models into the system while it is working. Then, assuming that the system has correctly anticipated possible changes, the requirements models can help to guide selection of new solutions. More recent work has tried to incorporate specific operators into the requirements model in order to add a measure of self-awareness to the requirements models. The separation between level 2 and level 3 of [Cheng et al., 2005] cannot be avoided, but level 2 might be greatly expanded. This is the approach of the RELAX framework [Whittle et al., 2009], Fuzzy Goals [Baresi et al., 2010], and the Markov models of [Epifani et al., 2009].

In the unanticipated, requirements evolution problem, there has been work in the area of search-based software engineering that is relevant to this chapter. That work focuses on combinatorial optimization, using evolutionary and local search algorithms to efficiently find solutions, e.g [Battiti et al., 2008; Finkelstein et al., 2009]. DDP [Feather and Cornford, 2003] supports quantitative reasoning and design selection over non-hierarchical requirements models, and the latest iteration uses search-based heuristics to find system designs Gay et al. [2009]. The principal difference is in the nature of the underlying model. Our framework uses goal models to provide for latent interactions between requirements that are satisfied. More recent work includes [Zhang et al., 2008], [Finkelstein et al., 2008], and [Zhang et al., 2007]. Again requirements models are fairly simple, and there is no notion of tradeoffs unless a numeric cost function has been assigned. While this flies in the face of most requirements processes, it may be all that can be obtained for prioritization, particularly if we refer to agile methods. Story points, for example, are a simple cost model that can be applied here.

A related variant is release planning (a.k.a. the Next Release Problem) which differs largely in the amount of time allotted to make a decision. In this thesis, I aim for solution finding in milliseconds, so that new systems can be quickly generated and brought online. In other work (for example, [Saliu and Ruhe, 2007], [Wohlin and Aurum, 2005], [Wnuk et al., 2009], [Durillo et al., 2010]) the focus is on presenting options to stakeholders for future product releases, often as part of a product line. The focus is on how to make a decision, where I focus on discovering the possible solutions which exist (and which may be inconsistent or unsatisfiable).

As I mentioned earlier, there is a continuum between self-adaptation and evolution when we consider the possible solutions might include service selection, and some measure of meta-reasoning in the requirements model itself (i.e., the model has a goal “Add new requirements”). In [Qureshi et al., 2010] we looked at a framework built on Techne for doing this, which was expanded in Qureshi et al. [2011]. The work of Inverardi and Mori [2010] has focused on the “foreseen” requirements, but describes a feature-oriented RE model that searches for new features to satisfy the changes in requirements.

Another class of approaches is described in Di Rosa et al. [2010]. Using a SAT solver fitted with preference relations, they can solve the requirements *selection* problem fairly quickly. Although there is no requirements framework described (it is targeted at AI planning) it is easy to see the applicability of an approach directly integrating preferences.

8.7 Conclusion: Evolving Requirements

The Requirements Evolution Problem (REP) is a common one. The definition is broad: any time a system cannot adapt itself to “unknown unknowns” (be they new requirements, newly available implementation components, or new domain properties), the designers are required to step in and make changes to the REKB. That is, acting to solve the REP exactly characterizes human-guided software maintenance that is not purely implementation-focused (i.e., maintenance that is not corrective). My position is that this maintenance ought to be done with reference to the REKB, since the REKB is where, properly, we can understand the interaction between user goals, domain constraints, and implementation. To that end, I have defined an operator, GET-MIN-CHANGE-TASK-ENTAILING, which can find, automatically, the optimal **new** solutions to select after revising the REKB.

Furthermore, this chapter has shown how the process of revising requirements problems is different than revising knowledge bases. The key objective of introducing the concept of “solutions” to the requirements problem is to insist that a revision either maintain or improve the state of our implementation. In knowledge representation we want our representation to closely reflect an agent’s beliefs about reality. In Requirements Engineering, on the other hand, we are interested in solving the requirements problem. That means that in certain circumstances we might reject new information if it does not improve on the old solution. The example I introduced (of a meeting scheduler) showed that if recency of information was paramount, it might lead to a possibly undesirable situation. I expect preference relations over (at least) the goals to resolve these revision problems.

This chapter has considered how to resolve evolving requirements problems. It has assumed that these changes do not produce an inconsistent REKB, but this assumption is constrictive. In the following chapter, I will tackle the case where our operations must act on an inconsistent REKB.

Chapter 9

Solving Inconsistent Requirements Problems

Managing conflicting information is a necessary component of the requirements problem. In the previous chapters the operations generated exceptions if an inconsistent state was reached during the reasoning. Conflicts express the case where one requirement cannot be satisfied simultaneously with another requirement. Conflicts are important issues in the design and maintenance of software systems. They help to guide implementations by restricting the set of possible solutions. In Requirements Engineering, however, conflicts (which produce inconsistency) need not lead to impasse, so a reasoner which cannot proceed is not a reasonable restriction.

As an example, let us assume that a company must demonstrate compliance with the PCI-DSS case study introduced in Chapter 5. The compliance procedure consists of numerous audits and tests. One requirement (R2.2.1) is that server functions (web server, mail server, file server, etc.) must be separated among machines. Another requirement (R3.5.1) says “Restrict access to cryptographic keys to the fewest number of custodians necessary.” If the customer tells the designer that it is not possible to purchase more than one server, which conflicts with R2.2.1, a classical system would pause until this inconsistent state was resolved, perhaps by permitting server virtualization, which is cheaper than purchasing new server hardware. However, this suggests a premature design decision. For one, it prevents the company from working on ways to ensure compliance with R3.5.1, which might involve working on the same servers that are the subject of R2.2.1. For another, it might be that the company decides to make a different strategic decision, using a third-party for mail servers, for example, and therefore passing R2.2.1.

As we saw in Chapter 6, solving the requirements problem is to find or show a specification of tasks

(T) and refinement information (R) such that

$$D \cup R \cup T \vdash G \tag{9.1}$$

where G are mandatory goals, while D refers to domain knowledge.

In order to do so, I define the following consistency criteria:

$$D \cup R \nvdash \perp \tag{9.2}$$

$$D \cup R \cup G \nvdash \perp \tag{9.3}$$

$$D \cup R \cup T \nvdash \perp \tag{9.4}$$

In that chapter, and Chapter 7, the operators generated exceptions if a call violated any of those criteria. In this chapter, I will introduce a different consequence judgement ($\mid\!\sim$), which was introduced for requirements problems in [Jureta et al., 2008] and elaborated in [Jureta et al., 2010]. This judgement is paraconsistent in the sense that it will permit us to continue to draw some reasonable conclusions in the presence of inconsistency in the REKB (i.e., not everything follows from an inconsistent set, and the set of conclusions may in fact always be consistent).

This chapter begins by introducing some approaches to accommodating paraconsistent reasoning in requirements. I then turn to the key operators from Chapter 6: ARE-GOALS-ACHIEVED-FROM, MINIMAL-GOAL-ACHIEVEMENT, GET-CANDIDATE-SOLUTIONS and GET-MIN-CHANGE-TASK-ENTAILING. I look at how they ought to be modified in the paraconsistent case. I then sketch a possible implementation for these paraconsistent operators based on the notion of "maximally consistent subsets". I use this notion to give a functional characterization of the three operators. I outline how this is implemented in the ATMS, and then consider related work on paraconsistent requirements.

9.1 Inconsistency and Conflict

It is important to properly characterize what is meant by the terms 'conflict' and 'inconsistency' in the context of the REKB. Chapter 9 will make extensive use of these concepts. As we saw in §2.2, in a formal language, such as a $\mathcal{L}$ above, inconsistency means that a theory in the language has derived or been told that a propositional variable is both True and False (in our case, that False is derivable). From this, in classical logic anything can be derived (*ex falso quodlibet*). The permissible formulas of the languages $\mathcal{L}$, however, prohibit this atomic form of inconsistency in the absence of formulas of the form $\alpha \to \perp$.

There are several ways to interpret the existence of a *conflict* relation between requirements A and B, recorded as $A \wedge B \to \perp$. The conflict might mean that neither requirement can be satisfied. The conflict could also mean that at most one, but not both can be satisfied. Finally, it could be more drastic, and suggest that the entire model must be resolved to remove the conflict. Techne adopts the second attitude. Part of searching for solutions to requirements problems is to find ways to ensure at most one of A or B, where A and B are in conflict, is satisfied.

In the requirements engineering research community, the term "conflict" has typically been used to denote social disagreement over the nature of the system requirements. Robinson et al. [2003] define it as "requirements held by two or more stakeholders that cause an inconsistency". The term "inconsistency" denotes the technical, formal existence of a "broken rule" [Easterbrook and Nuseibeh, 1995]. Zowghi and Gervasi [2003] show that "consistency" is causally related to requirements "completeness": a more complete requirements document is often less consistent (since more competing requirements are introduced). In Techne, the conflict relation is formally between two requirements, and not between stakeholders (some would insist that inconsistency is more properly between two *or more* requirements). Techne assumes that the requirements problem it is modeling is canonical. Any conflicts between stakeholders, such as a disagreement over terminology, are the purview of other techniques (e.g., model merging). For the REKB, conflict is the presence of a rule with $\perp$ present (e.g., $A \wedge B \rightarrow \perp$), while inconsistency is when false ($\perp$) is derived. Ultimately, we not only want to permit conflict (and possibly inconsistency) to be represented; we also want to define what we ought to do when inconsistency is detected.

Let us consider an example. As designers we have a requirements problem for designing software for a new intravenous fluid (IV) pump. These pumps are machines with onboard, embedded software. They are used to periodically infuse a patient with medication on a very strict schedule. The problem is safety-critical because failure of the pump can mean death for the patient. A good example of conflicting requirements comes from the case of failed Baxter IV pumps, which spawned the Food and Drug Administration in the U.S. to begin a pump safety program[1].

On the one hand, we would like to support battery backup, because batteries are mobile and batteries will not fail when the hospital loses power. On the other hand, we do not want batteries, because they can do poorly in the hospital environment, can explode and will not fail noisily (i.e., they will fail without warning us to change them). If our problem says we **must** do both "use battery" and "avoid using battery", then we have conflicting requirements. A useful language for the requirements problem needs to highlight that both are mandatory and that the two are conflicting. In a classical language, this state would alert us that we cannot reason further without resolving this conflict. In previous chapters

[1]See the FDA website: http://goo.gl/3k4Db

I assumed that when inconsistency is detected, the REKB generates an exception, and further reasoning stops. However, this is not satisfactory in the general case, so this chapter considers paraconsistent operators on the REKB, which tolerate inconsistencies in the REKB.

Classical languages, such as propositional logic and first-order logic, cannot tolerate inconsistency, in the sense that no useful reasoning can be done in its presence, and yet, in the requirements engineering domain, tolerating inconsistency is important. Nuseibeh et al. [2001] give a few important reasons:

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

Perhaps the most useful reason for the case of evolving requirements problems is the second. Avoiding premature commitment, in the sense of Thimbleby [1988], means to wait until the 'last responsible moment' to make decisions regarding the system. Not only does this apply to deciding *how* to satisfy our goals, but also in the choice of those goals themselves. Tolerating inconsistency therefore allows us to continue to make progress on design (and even implementation) while firewalling the conflicting parts of the system.

Tolerating inconsistency is also known as *paraconsistent reasoning*. A paraconsistent language, broadly, is one which does not trivialize in the presence of inconsistency. In this chapter, and in Chapter 7, my operations will insist that these conflicts not exist (classical consistency). However, since this is an unrealistic assumption for requirements problems, Chapter 9 will show how we can re-define these operations to be paraconsistent: that is, to continue to give meaningful answers even when conflicts exist.

There are two important questions for paraconsistent reasoning in requirements engineering:

1. What does a representation of the requirements problem look like?
2. What does it mean for the representation to be consistent or inconsistent?

To answer question 1 for my REKB approach, I first identify some interesting sorts on theories in the REKB. These theories can be for domain knowledge D, i.e., members of DAS, one interesting part of which is refinement information, in the form of implications and conflicts, which I will call R. These theories can also contain specifications of tasks T and mandatory goals G. Logical inconsistency can occur when a conflict (of the form $A \wedge B \rightarrow \bot$) is in R and $\bot$ can be derived from the union of T, D and G. Then solving the requirements problem is to find or show a T such that the following holds:

$$D \cup R \cup T \vdash G \tag{9.5}$$

Question 2 asks what it means to be consistent. Consider the following consistency criteria:

$$D \cup R \nvdash \bot \tag{9.6}$$

$$D \cup R \cup G \nvdash \bot \tag{9.7}$$

$$D \cup R \cup T \nvdash \bot \tag{9.8}$$

Then consistency in the REKB means at least criterion (9.6) holds. In ongoing work with Ivan Jureta and Alex Borgida, we call these 'blockers': since domain assumptions are true universally, then no solution can ever exist if criterion (9.6) does not hold. We also want to insist that all mandatory goals are achievable together (i.e., $\bigwedge G$), criterion (9.7). Finally, we should insist that not only can we achieve those goals, but that there are consistent tasks which will do so, criterion (9.8). Note that it is not a problem if there are non-desired goals which are unsatisfied, i.e., not in G.

In some requirements problems we may wish to speculate about certain low-level atoms being true, i.e., "suppose goal g_1 were achieved; what else is necessary to achieve top-level goal g_0?". In that case we may *hypothetically* assert these goals by introducing a "refinement" ("R: goal g_1 is achieved") which *assumes* that it has been achieved.

The operator specifications below follow the above discussion of inconsistency, by alerting the user if conditions 9.7 or 9.8 are violated. Classically, this is a signal that the inconsistency must be removed. Paraconsistently, for the reasons presented by Nuseibeh et al. [2001], reasoning should continue. We remain classically consistent for now, i.e., throw exceptions. The remainder of this chapter, and the two following, consider the requirements problem only in the classical case. In Chapter 9 I return to the problem of tolerating inconsistent requirements.

9.2 Paraconsistency in Requirements Engineering

As I discussed, there is the issue of defining inconsistency in a language, and then there is the matter of what to do with this inconsistency. In this chapter I take the approach the we wish to avoid trivial conclusions, as happens when we use the classical consequence operator $\vdash$. Inconsistency tolerance in a language is known as paraconsistent reasoning. As we saw in §9.1, there are good reasons for supporting paraconsistency. Hunter [1998] defines four approaches for reasoning paraconsistently:

1. A classical language with a subset of classical proof theory ("weakly-negative logic");
2. A subset of classical language, a subset of classical proof theory, and four-valued semantics ("four valued logic");
3. Rewrite data and queries and use clause finding ("quasi-classical logic")
4. Reason with consistent subsets of the problem ("argumentative logic").

The second category closely resembles the approach behind [Sebastiani et al., 2004]. That paper defined simple predicates for declaring requirements "Satisfied", "Denied", and partial versions thereof. Since a requirement can obtain any combination of "True" and "False", there are four values possible. The last category most closely resembles the paraconsistent language Techne uses, as defined by [Jureta et al., 2010].

I will present a general approach to defining what it means to draw conclusions from a possibly inconsistent set of assumptions, denoted by the $\mid\!\sim$ symbol. I will then define paraconsistent versions of the operators from Chapter 6 using $\mid\!\sim$. In the classical definition, we generated exceptions in the case where a consistency criterion was violated, i.e., whenever it was possible to classically derive $\perp$. In the paraconsistent case we continue to derive conclusions. We note the presence of the inconsistency, but continue regardless.

To properly define $\mid\!\sim$, I first leverage the definitions used in the argumentative logic defined in [Elvang-Gø ransson and Hunter, 1995] (EGH). EGH defined an argumentative logic based on the concept of maximal subsets of a logical theory Δ. In particular they defined the following important subsets:

$$\mathsf{CON}(\Delta) = \{\Pi \subseteq \Delta \mid \Pi \nvdash \perp\} \tag{9.9}$$

$$\mathsf{MC}(\Delta) = \{\Pi \in \mathsf{CON}(\Delta) \mid \forall \Phi \in \mathsf{CON}(\Delta),\ \Pi \not\subset \Phi\} \tag{9.10}$$

That is, CON defines a consistent set as one which is not (classically) inconsistent; MC identifies sets that are consistent and have no consistent supersets. Note that if Δ is consistent, then MC(Δ) = Δ.

This leads to the following definition:

Definition 1. $\Delta \mid\!\sim S$ *iff*

1. $\exists \Pi \subset \Delta$,
2. $\Pi \in \mathsf{MC}(\Delta)$,
3. Π *contains all implications in* Δ, *(written Implications(Δ)),*
4. $\Pi \vdash S$

The first obvious criterion is that in case that Δ is classically consistent, we would like $\vdash$ and $\mid\!\sim$ to behave identically. The motivation for condition 3 is specific to Requirements Engineering, particularly using the Techne family of languages and their methodology: in addition to atoms, the only other formulas allowed by Techne are Horn implications; these encode domain knowledge, refinement information (e.g., from goals to subgoals or tasks) and conflicts. Since in the specification of operators the only other thing added that may become inconsistent are atomic tasks/goals, the rules seem to us to be more central (have higher priority) and since a set of Horn implications is known to always be consistent[2], condition 3 above does not affect the existence of maximal consistent sets.

I am aware that the above definition is "credulous" since it depends only on the existence of *one* set Π, from which S can be derived. In contrast, much of non-monotonic reasoning and database reasoning with inconsistent data deals with the "skeptical" mode: S must be derivable from *all* maximally consistent Π of the above form. I defend my choice later.

9.3 Paraconsistent **REKB** Operators

From the definition of paraconsistent consequence ($\mid\!\sim$), the operators follow, as before, with the difference that they use that $\mid\!\sim$ consequence relation, and the exceptions generated are changed. As well, if the arguments to the operation are themselves (internally) inconsistent, the reasoner may still generate an exception.

Operation 8 — PARACONSIST-ARE-GOALS-ACHIEVED-FROM

Domain: REKB $\times$ assumeT :$\wp$(GOALS $\cup$ TASKS) $\times$ concludeG :$\wp$(GOALS)

Co-Domain: Boolean

Effect: returns true iff REKB.TH $\cup$ assumeT $\mid\!\sim$ $\bigwedge$concludeG

Throws: Only throws exception if concludeG $\cup$ Implications(REKB.TH) is inconsistent.

Δ from Equation 1 is now the union of the information in REKB.TH and the assumeT. The operation returns True if there is at least one subset Π of REKB.TH $\cup$ assumeT which entails concludeG according to the definition of $\mid\!\sim$. We continue to signal problems with inconsistent goals so that the user can tell when their non-achievement is intrinsic from their internal conflicts, in contrast to absence of sufficient refinement information or tasks. But the set of actions may contain inconsistencies because $\mid\!\sim$ assures only consistent subsets are selected.

[2]Just assign False to all antecedents; since there is no negation, the result will be a satisfying valuation.

As I remarked earlier, the definition of paraconsistent reasoning used here is "credulous", in that the set concludeG is deduced if there is *at least* one consistent set that can derive it (and contains the implications). The other approach is skeptical, where I would only accept the truth of concludeG in the event that *all* maximally consistent sets could derive it. The distinction between credulous and skeptical is important in knowledge representation, but I argue that it is less significant in requirements problems: since we are considering possible future states of the world, we are making assumptions about which tasks to implement. We can easily discard those tasks, and the requirements they imply. Implicitly what the skeptical semantics for paraconsistency in requirements engineering do is restrict the nature of the eventual system we build. The only constraint imposed is that implications may not be discarded.

Example 9. *Consider the case where classical ARE-GOALS-ACHIEVED-FROM would generate an exception. This occurs, for one, if the set assumeT is not consistent with the set resulting from $\bigwedge$ concludeG. Consider the sets* REKB.$TH = \mathbf{R} \cup \mathbf{D}$ *with* $\mathbf{D} = \{\}$, $\mathbf{R} = \{B \wedge C \rightarrow \bot, A \rightarrow C, B \rightarrow D, C \rightarrow H, D \rightarrow H\}$, $\mathbf{T} = \{A, B\}$, $\mathbf{G} = \{H\}$. *A call to PARACONSIST-ARE-GOALS-ACHIEVED-FROM($\mathbf{T}, \mathbf{G}$) produces the answer True, because goal H can be achieved using a maximal consistent subset that contains A or one which contains B. Note that in the classical case parameter $\mathbf{T}$ would have generated an exception, though not any subset of it.* ■

Operation 9 — PARACONSIST-MIN-GOAL-ACHIEVEMENT

Domain: REKB × concludeG :$\wp$(GOALS)

Co-Domain: $\wp(\wp(\text{TASKS}))$

Effect: returns a set that contains all sets S of tasks such that REKB.TH $\cup$ S $\mid\!\sim$ $\bigwedge$concludeG, and no subset of S has this property.

Throws: exception if concludeG $\cup$ Implications(REKB.TH) $\vdash \bot$.

In the consistent case, we saw that this is an abductive search. Abduction normally only works from consistent theories, so the MINIMAL-GOAL-ACHIEVEMENT operation generates an exception if the theory (REKB.TH) is not consistent with concludeG. The same exception is preserved in the paraconsistent version, but now we can continue to reason if the REKB.TH is not internally consistent.

Example 10. *Consider the REKB represented in Fig. 9.1, which can be represented as* $\mathbf{R} = \{D \wedge E \rightarrow B, F \wedge H \rightarrow C, C \rightarrow A, B \rightarrow A, E \wedge F \rightarrow \bot\}, \mathbf{D} = \{E, F\}$. *If we then define* $\mathbf{G}$ = *concludeG* = $\{A\}$, *the problem is classically inconsistent since there is a conflict when we identify potential solution sets which must contain the domain assumptions E and F. Paraconsistently, however, we can identify two separate answers to the operation:* $\mathbf{S_1} = \{D, E\}, \mathbf{S_2} = \{F, H\}$. *This supports our desire to continuing to reason*

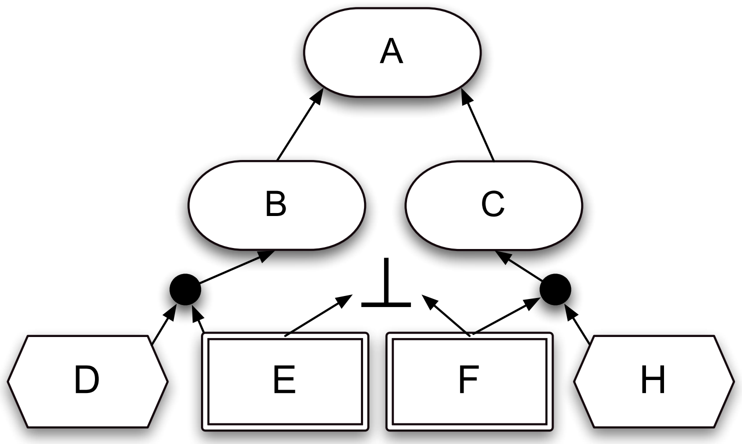


Figure 9.1: The REKB from Ex. 10. The double lines for E and F indicate they are asserted.

despite a conflict. This example mimics the example at the start of the chapter: we have assumptions that we may not buy more servers, but also that non-compliance with the PCI-DSS will be expensive. ■

Operation 10 — PARACONSIST-GET-CANDIDATE-SOLUTIONS

Domain: REKB × mandG : ℘(GOALS) × wishG :℘(GOALS)

Co-Domain: maxC:℘(⟨solnT, satG⟩) where solnT :℘(TASKS) and satG :℘(GOALS)

Effect: maxC is a set of tuples such that 1) REKB.TH ∪ solnT $\mid\!\sim$ satG, 2) satG has all goals in mandG (which contains those goals for which the REKB has been told ASSERT-ATTITUDE(*mandatory*)), and as many goals in wishG as possible, 3) solnT is subset minimal to achieve the above.

Throws: throws exception if mandG is inconsistent with Implications(REKB.TH).

Operation PARACONSIST-GET-CANDIDATE-SOLUTIONS now only requires that the set solnT paraconsistently derive satG, i.e., there is a consistent subset of REKB.TH from which one can classically prove satG, using solnT.

Operation 11 — PARACONSIST-GET-MIN-CHANGE-TASK-ENTAILING

Domain: REKB × goalsG :℘(GOALS) × originalSoln :℘(TASKS)× DIST-FN

Co-Domain: ℘(℘(TASKS))

Effect: Return the set of task sets S which solve REKB.TH ∪ S $\mid\!\sim$ goalsG, and which minimize DIST-FN(S,originalSoln).

The PARACONSIST-GET-MIN-CHANGE-TASK-ENTAILING operation is largely unchanged from the classical version, since the focus remains on the DIST-FN operation comparing new and old solutions. Since we operate on consistent subsets, the caveat is that a single new conflict relation could produce a dramatic difference in the available solutions (as we would expect).

The paraconsistent versions of the operators have illustrated how paraconsistency is a fairly natural concept in solving the requirements problem. Our focus remains on the appropriate sets to return, but we adopt a credulous approach in which a single consistent subset of the larger REKB can be used to derive our mandatory requirements. This model of operation also maps nicely to our choice of the ATMS for implementation, as I discuss in the next section.

9.4 Implementing Paraconsistency in the ATMS

The implementation for the paraconsistent operators is the same as in §6.3.2, except for the exceptions detected, since the ATMS (assumption-based truth maintenance system) approach natively supports the semantics I defined in the previous section. In §7.1.1 I introduced the properties of an ATMS. The ATMS has built-in support for paraconsistent reasoning; this was one of the major design decisions separating it from Doyle [1979]'s Justification-based Truth Maintenance System (JTMS) that preceded it. In the JTMS, if one added a justification from an antecedent to a NOGOOD state then the JTMS raised an exception, exactly as the classical version of the REKB operators does.

The ATMS, by comparison, simply eliminates any explanations (minimal sets) which contain inconsistency. Such a scenario exists in Ex. 10 from before. E and F are in conflict. In ATMS terminology, we say that their explanations form a NOGOOD set (the explanations for assumptions are the assumptions themselves). Thus, the set $\{E, F\}$ is added to the NOGOOD table. However, since there are two justifications for A, namely B or C, not all explanations for A are subsumed by a NOGOOD, and hence, we can conclude that A is explained by both $\{D, E\}$ and $\{F, H\}$.

Therefore, in implementing PARACONSIST-ARE-GOALS-ACHIEVED-FROM, for example, I mark the nodes in assumeT as assumptions (thus 'asserting' them), and then use Alg. 3, which simulates the case where a dummy node is implied by the nodes in concludeG. If there is a minimal explanation for $\bigwedge$ concludeG which is equal or is subsumed by assumeT, then we can answer True. The ATMS has removed inconsistent explanations from the possible explanations for concludeG. A similar approach is followed for the remaining algorithms.

This is quite important for solving requirements problems. Let us consider the example of a secured entrance access system from Zowghi and Gervasi [2003]. Their paper focuses on ensuring that the specification is consistent as the requirements are refined. In the REKB approach, the system would consist of R, the implications and conflicts between the requirements (e.g., $\mathsf{openGate} \rightarrow \mathsf{canEnter(p)}$); G, the high-level objective of restricting entry; and T, specifics of accomplishing that goal. Domain assumptions are not explicitly mentioned in their paper, but in addition to the implications, might capture the fact that the gate never fails, that people never lose ID cards, etc. Then one question we might pose to the REKB is "What tasks must the organization carry out to ensure G is satisfied", which is the MINIMAL-GOAL-ACHIEVEMENT operation.

Let us then introduce the changing business objective of "permitting unauthorized access in the event of a fire", a new member of G, along with the hypothetical scenario that there is a fire. This conflicts with the initial specification, since we are now trying to prevent unauthorized access, and allowing it in this specific scenario. In [Zowghi and Gervasi, 2003], the solution is to revise immediately. As I mentioned, however, there are times when this constant revision to remove inconsistency is not necessary. Consider the case when we are doing iterative development of the company-wide security system, of which the access door is but one component. That there is a conflict between the need to permit unauthorized entry and the need to prohibit unauthorized entry is immaterial in developing, for example, a system to ensure company computers are password protected. In that case, we can tell the REKB that the second goal is not-mandatory, and even though there is a logical inconsistency, continue to reason about the satisfaction of the remaining mandatory goals.

The operation PARACONSIST-GET-CANDIDATE-SOLUTIONS has a slight change from the previous implementation. Recall that our REKB will be composed of domain assumptions and implications. Since PARACONSIST-GET-CANDIDATE-SOLUTIONS is seeking to maximize satG while minimizing solnT, we should remove from the returned subsets those members which are sort DAS. This is in order to work solely with minimal sets of tasks, according to the operation definition. However, it is conceivable that some of these sets will contain minimal subsets of tasks which (together with the domain assumptions) subsume other solution sets. Ideally, we would seek to maximize the number of domain assumptions in a set which maximized satG, before considering the tasks themselves. This is because domain assumptions represent commitments we have already made, or which already exist, and presumably reduce the work involved. This is not something which the ATMS can do, but rather, must be implemented in the problem solver.

9.5 Complexity Considerations

Since I defined $\mid\!\sim$ using maximal consistent subsets (which is typical of the argumentative logic approach), the complexity becomes that of non-monotonic reasoning, i.e., NP-hard, even for the previously linear PARACONSIST-ARE-GOALS-ACHIEVED-FROM operation. This is because of the extra effort involved in calculating the maximal sets.

For the PARACONSIST-GET-CANDIDATE-SOLUTIONS operation, we now have the extra work of filtering our returned sets to remove domain assumptions, which is linear in the number of returned solution sets. Furthermore, if we wish to work with domain assumptions, we will need a second linear check to consider the maximal number of domain assumption, and thus the entire operation can be $O(N^2)$.

9.6 Other Treatments of Inconsistency in Requirements

Requirements conflicts and inconsistency have been studied in depth in the requirements engineering domain. In [Jureta et al., 2011], a paper in progress, we provide a detailed survey of this research. This section will focus on the more salient approaches. I begin by considering how others have defined conflict or inconsistency in RE. Following that I will examine how that research has approached working with conflict. One approach is to tolerate it; the other approach is to discover it for future repair.

9.6.1 Default Reasoning

An Australian research programme looked at the notion of evolving requirements, captured in [Zowghi and Offen, 1997; Antoniou, 1998; Ghose, 2000]. This work was inspired in part by the Viewpoints research of [Easterbrook and Nuseibeh, 1995], and focused on tools and techniques for managing inconsistencies in developing a requirements model. They applied non-monotonic reasoning in the form of the logic of defaults [Poole, 1988] to requirements. Using a default logic supports reasoning with assumptions and multiple (inconsistent) problem formulations. Default logic is non-monotonic in that TOLD facts can later be contradicted and no longer concluded; for example, the sentence “goal G is refined by task T” can be over-ruled if new information is discovered that says “goal G has no refinements” (for example). In classical logic, the original sentence cannot be retracted.

Zowghi and Offen [1997] use Poole’s default logic for handling inconsistency in evolving requirements. Default logic supports the notion of partial specification by allowing an analyst to state that it is *assumed* that something is the case, in the absence of information to the contrary. Default logic is at the same level of the polynomial hierarchy as abduction (Σ_2^P), so the complexity of the reasoning is the same as in the REKB approach. Poole [1990] has characterized the difference between abduction and default reasoning as the difference between seeking explanations and seeking predictions. In abduction, one starts with an *explanandum*, the observation one seeks to justify using (sets of) explanations. In default reasoning, one starts with a theory, and seeks to predict the value of the explanandum, which is unknown. In the REKB, the explanandum is the (set of) mandatory goals.

Zowghi and Offen, by contrast, approach things from a verification perspective. Their central concern is to ensure that the requirements specification is complete and consistent. In that context, it makes sense to use prediction: given the current state of the system (with respect to the assumptions used in the defaults), will the mandatory goals be true? The REKB, on the other hand, is concerned with the requirements problem. The central task is to find solutions to that problem. Failing to find any solutions implies that the problem is incompletely or improperly formulated.

In both cases, algorithms for solving the reasoning task are exponential in running time. However, the choice of default or abductive reasoning does seem to imply some methodological tradeoffs. The approach I defined for the PARACONSIST-ARE-GOALS-ACHIEVED-FROM operator is closest to the default logic of Zowghi and Offen, since it can be used to do scenario exploration by testing multiple sets of tasks.

To evolve the specification, Zowghi and Offen use a revision function over extensions (maximally consistent subsets), incorporating the AGM postulates to define how revisions should be undertaken, and partially ordered using epistemic entrenchment for choosing between wffs in the theory. In the REKB, revising to ensure consistency is not necessary. The only question that is relevant after a revision is whether the requirements problem can still be satisfied.

Closely aligned with this perspective, and perhaps closest to our view of a REKB, is the work of Ghose [2000], in the REFORM framework. In this paper, Ghose identifies three main properties a system for managing evolution ought to have:

1. Distinguish between what he calls *essential* and *tentative* requirements (my mandatory and preferred 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.

The first two are shared with the REKB, while the notion of decision criteria for selecting a solution, discussed in Chapter 7.2, essentially covers the third point. The criteria that Ghose states a framework should satisfy are likewise similar to the REKB approach:

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.

Again, the REKB approach is closely related: it also focuses on minimal changes, and the version control operations specify in detail how to implement point 4. Point 3 is interesting. Ghose insists on obtaining from an oracle the possible critical states of system behaviour, which he calls a trajectory. With this predictive oracle, the solutions to the requirements problem captured in his language can then be optimized. The oracle is capturing significant contextual variation in the assumptions. This is *anticipated* change; if a situation occurs that is not part of the trajectory, the problem must be re-reasoned. The advantage is of course that this permits Ghose to do "look ahead" and solve (for small trajectories) the best set of decisions to make. In the REKB approach, by contrast, I am not assuming anything about the future state of goals, tasks or domain assumptions. This is mainly a philosophical difference: it is certainly true that scenario planning is possible, and perhaps useful for a short sequence of states; however, work in planning, for example [Boutilier et al., 1996], has recognized that anticipating future changes is futile. This viewpoint also characterizes agile software development, and also motivates the focus on unanticipated changes in my thesis.

Ghose's approach to evolution is similar to the notions I cover in the GET-MIN-CHANGE-TASK-ENTAILING operator. He discusses additions and subtractions from a requirements representation. The key approach is to select a suitable revision choice operator in order to choose whether an addition ought to be selected. This close parallel between the REKB and REFORM approaches is positive, as it shows that these issues are commonly seen as important. However, it raises the question of what is new about the REKB approach (some eleven years later). The novel characteristics of the REKB approach are:

1. The use of the ATMS as a theorem prover, based on Horn propositions, which is decidable, rather than a first-order prover.

2. A focus on abduction for the semantics of the minimal achievement operator, rather than default logic. In particular, this implies a focus on minimal sets of implementation tasks which satisfy the mandatory ("essential", according to Ghose) requirements.

3. The decision to focus on propositional Horn logic as a representation language; for one thing, this allows the ARE-GOALS-ACHIEVED-FROM operation to be linear. Secondly, propositions are more suited to early 'design' requirements engineering than first-order languages.

4. The incorporation of the CORE ontology to separate between domain assumptions, implementation tasks, and goals.

5. Support for reasoning incrementally.

6. Abandonment of the ideal of anticipating all future critical scenarios in favour of expecting the unanticipated.

While [Ghose, 2000] makes reference to proofs and implementation, it would appear this no longer exists[3]. This makes it hard to conclude anything about the tractability of the approach.

9.6.2 Identifying and Characterizing Conflict Relations

The issue of conflicting stakeholder requirements was the subject of the Viewpoints project [Easterbrook and Nuseibeh, 1995]. This line of research has continued as an examination of merging models [Sabetzadeh et al., 2007] and managing multiple sets of models [Salay et al., 2009]. The REKB could leverage this work in order to combine and manage multiple requirements problems.

van Lamsweerde et al. [1998] considered technical inconsistency in great detail. They first listed several types of descriptions one might create for requirements problems, using the KAOS modeling language. A description type might be a set of goals for the system-to-be or it might be a set of viewpoints on the system. Depending on the description type, they said that a "set of descriptions is inconsistent if there is no way to satisfy those descriptions all together. " The paper then listed nine types of inconsistency:

1. Process-level deviation: inconsistency between the metamodel and the model, where the metamodel is a process model for modeling requirements problems.

2. Instance-level deviation: inconsistency between the model and instances of the model, such as specific tasks.

3. Terminology clash: a syntactic inconsistency commonly found in multiple viewpoints models.

4. Designation clash: semantic inconsistency, where a concept in the model means different things to different people.

5. Structure clash: inconsistency in the representation of a domain concept in the model.

6. Conflict: domain assumptions A_i are minimally inconsistent with one another's conjunction. This is the conflict relation in Techne.

7. Divergence: a Conflict that is time-sensitive. Requires the use of boundary condition definitions, used to define the particular set of circumstances under which this would be divergent.

8. Competition: a Divergence with only one affected goal.

[3]source: personal communication

9. Obstruction: a single goal is obstructed in specific boundary conditions.

These do not all have analogues in Techne, because KAOS is more expressive for requirements scenarios (e.g., at time T goal G must be satisfied). Similarly, van Lamsweerde and Letier [2000] introduced the concept of requirements obstacles, divergences that are undesirable states of affairs. These are separate types of requirements (anti-goals) in the model. Eliciting the presence of these obstacles allows the modeler to introduce techniques to avoid them. This is a form of anticipated system adaptation.

The approach perhaps furthest from the formal techniques described before is to offload this problem onto the stakeholder. In the i* goal-oriented requirements framework, conflict is reconciled during the model evaluation, using an interactive tool [Horkoff and Yu, 2010]. Users are prompted to understand and remove or tolerate the conflict in the goals they have previously modelled. This is clearly helpful in reconciling potential inconsistency. However, users must understand the reasons for the conflict, and users must be available during the model evaluation. Because this framework extends the Satisfaction/ Denial predicate approach of [Sebastiani et al., 2004], inconsistency is tolerated, since it only exists outside of the formal model (e.g., when a goal is both Fully Satisfied and Fully Denied). However, this raises the problem of 'conflict pollution': the semantics of the reasoning imply that a single conflict can quickly propagate this status throughout the model through AND decompositions. A model which consists mostly of conflicted requirements is not useful.

9.6.3 Logics for Managing Inconsistency in RE

Some RE languages and frameworks work with inconsistency. One approach is to define additional truth-values for elements in the language. Standard propositional logic is two-valued (True,False), but one could add the values Unknown, or Unknown and Disputed, to create a three or four-valued logic, respectively. This type of logic is usually represented with two orders over the possible values for a given propositional variable, as shown in Fig. 9.2. The truth-order reflects the difference between True and False. The knowledge order reflects how much information we have about the given atom. Too much information results in the Disputed valuation.

Reasoning with four-valued logics is more complex; the proof theory is not as well-understood as standard proof-theoretic semantics for two values. We created a simple four-valued requirements reasoner using Belnap [1977]'s logic, which we called **L4**. We developed a system for reasoning over this logic [Jureta et al., 2009a], but ultimately we could not reconcile what we wanted to express with what the logic could express. Consider a goal with a value of Unknown (indicating we do not know its true value), and a second goal which has a value of Disputed/Both (indicating we have conflicting information about

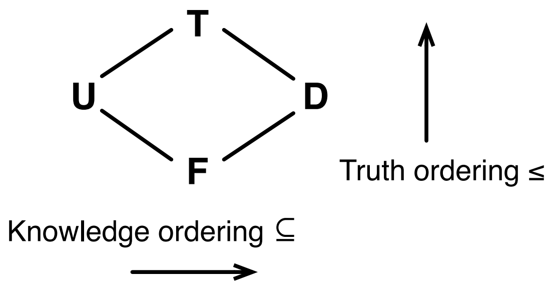


Figure 9.2: Lattice for Belnap's L4

its value). What value do we assign to a goal they jointly refine? The challenge is to come up with a rule for this relation that produces the results we would expect, every time. This is trivial in the simple case, but grows more complex with the application of examples. This is indeed what we concluded after implementing the **L4** logic for reasoning on requirements problems: the results we obtained did not match what we expected. For example, our reasoner would tell us that a goal was Unknown when we thought it ought to be True.

Chechik and Easterbrook [2001] use a four-valued logic to do viewpoint merging. The logic is more appropriate in this context, because we need to understand both the information about a given node as well as its truth or falsity.

9.6.4 Finding Inconsistency

Research in requirements engineering has looked at using inconsistency as diagnosis. The presence of inconsistency highlights a problem that must be resolved. This is particularly important when doing requirements elicitation, as the presence of inconsistency represents a conflict between points of view, or a misunderstanding of the domain. In either case, resolving the inconsistency guides the repair activity. This might be called a "theory-building" approach to RE.

Menzies et al. [2000] use abduction to unify inconsistent Viewpoints [Easterbrook and Nuseibeh, 1996]. Viewpoints allow each user to build their own theory of the world; inconsistencies arise when there are differences in worldviews. This is the theory-building approach to RE; by contrast, the CORE ontology takes a problem-solving approach: first, define the problem (that is, build a theory), and then use that model to find possible solutions. In other words for CORE it is not sufficient to have a model that is valid; we also want to find (feasible) solutions. Theory building asks requirements engineers to test or refute the given model, that is, to find counter-examples to elements and relations in the model.

Russo and Nuseibeh [1999] wrote about using QCL and abduction to find resolutions to inconsistencies. The abductive reasoning procedure produces a set of possible repairs to the inconsistent model. They give as an example the case where a library both has a book available for lending, and has lent it out. In this case the inconsistency is that the book is available and is borrowed. (The inconsistency here would seem simple enough to remove using a sequential logic). Abduction is used to reason from the background knowledge (the state of the world, e.g., BookAvailable) to produce diagnoses /repairs for the inconsistency. Abduction is not used to determine solutions to the model.

Chapter 10

Conclusion

10.1 Summary and Contributions

This thesis introduced the REKB as a way to understand what actions to take when a software system evolves. I have argued that while it is important to accommodate change, merely *reacting* is not sufficient; one must also take appropriate actions. To do this, I adopted the Requirements Problem approach first introduced by Zave and Jackson [1997], and refined by Jureta et al. [2008]. The Requirements Problem, simply stated, says that the initial step in software development is to find specifications of an implementation which will satisfy the requirements, with respect to some domain properties.

In Chapter 4 I introduced three empirical studies I conducted which explored how changing requirements drove changes in implementations. This provided support for the focus on the Requirements Problem as an important framing device for software maintenance activities. This motivated the introduction of the requirements engineering knowledge base (REKB) in Chapter 6 , a functional specification of the important operations in a requirements engineering perspective on software evolution. In particular, I defined an operation that helped find and choose solutions to the Requirements Problem.

The following chapters (Chapters 7 and 8) described one particular implementation of that functional specification. I conducted empirical studies which supported the use of the REKB as a knowledge base for taking maintenance and design decisions.

In general there are several dimensions in a requirements engineering approach to software evolution. There is the dual nature of the dynamicity of the problem (whether it changes or not). In addition, we ought to consider whether we are searching for a solution (a search problem) or trying to decide between those solutions (a decision problem).

Chapter 9 expanded the REKB to the case where we wish to reason paraconsistently. This is the case if conflicting requirements should not prevent us from continuing to draw conclusions from the REKB. At the close of that chapter, the REKB is a functional specification that can manage changing and possibly inconsistent requirements problems. It is therefore a powerful approach to the challenge of understanding how and why to maintain software systems. It represents an improvement over previous approaches to requirements modeling and representation in supporting inconsistent models, providing a well-defined set of operations on those models, implemented in a scalable tool.

10.2 Future work

There are three areas which merit further study. These are handling large-scale requirements models, understanding more about the use of requirements, and expanding on the nature of decision-making in the REKB.

It is not my intent to argue that the REKB approach can model all complex requirements problems. As Hopkins and Jenkins [2008] makes clear, for large systems there are simply too many competing views on the problem that each need their own perspective. Rather, the REKB should be seen as the way to deal with the challenge of identifying minimal and optimal solutions to the requirements problem the REKB defines. Developing the REKB, assigning tasks, and identifying domain assumptions are each properly handled with other tools. Requirements at large-scales, as defined in [Regnell et al., 2008], are very similar to any other large complex space: we will encounter problems understanding what is going on, and in particular, where and how to make a change.

One omission in my research has been the social side of socio-technical systems. Changing requirements are caused by some external factors, and often these factors are driven by social interactions [Lyytinen and Newman, 2008]. Furthermore, acting on information about what solutions to choose is a social endeavour as well. This thesis would do well to expand itself to explain requirements change in the context of socio-technical systems change. How are decisions being made? What impact do social factors have on changes in requirements? A key way to understand the answers to these questions will be to conduct surveys and studies that can situate the technical problem (how to manage changing requirements) within the social context.

The final aspect that is not well-developed in this work is the notion of optimization and decision-making. These topics have been well-studied in other disciplines such as economics, for example, in multi-objective optimization, linear programming, and decision theory. These seem very relevant for software maintenance of evolving systems. How do we select solutions to requirements problems? A solution is some configuration of tasks which will achieve the requirements. Since our requirements models permit alternatives, we will almost certainly have many possible solutions. Which solution is optimal? How is optimal defined?

10.3 Driving Software Maintenance

There is a focus in both academic and industrial software engineering on greenfield development, when most systems are legacy or brownfield [Boehm, 2009]. Furthermore, "All systems are legacy systems. If not the moment they are implemented, then just wait a few minutes.[1]" This implies, as Lehman identified in [Belady and Lehman, 1976], that we must constantly take corrective action to ensure that these legacy systems meet the requirements set out for them. This is even more the case when software does not exist in isolation, but interacts with many other systems, software, hardware, and human. To reinforce this perspective, we have the statement from Hopkins and Jenkins [2008] that "Today's IT architects should regard themselves as Brownfield redevelopers first, and exciting and visionary architects second."

One approach to understanding the activities involved in this corrective action, and in all software design, implementation and maintenance, is as a theory-building exercise [Naur, 1985]. Naur believes that the true activity in building software is to construct a shared theory among the developers. Producing software is instantiation of the theory. This explains why it is difficult to adopt another organization's code base, or to understand legacy code: a new theory must be developed in the minds of the developers. My work argues that this theory development can be thought of as formulating the nature of the requirements problem for that context.

With respect to modification (maintenance and evolution) of software, Naur comments that:

> In a certain sense, there can be no question of a theory modification, only of a program modification ... the problems of program modification arise from acting on the assumption that programming consists of program text production [...] The decay of a program text as a result of modifications made by programmers without a proper grasp of the underlying theory becomes understandable. [] For a program to maintain its quality it is mandatory that each modification is firmly grounded in the theory of it [Naur, 1985, p. 257].

I do not agree that theories are never modified – indeed, they must be modified to even exist in the first place. Furthermore, it is often in trying to express a theory, that is, build a system, that the theory is refined. However, with respect to his statement that problems in software design come

[1] http://jeffjonas.typepad.com/jeff_jonas/2010/12/re-hosting-legacy-systems-ground-hog-day.html

from assuming that design is about code creation, the arguments in this thesis support this. If software production is based on theory creation, it would seem foolish to allow that theory to remain tacit. My thesis has demonstrated that the REKB can capture this theory. No physical representation can hope to capture *all* the subtleties of a mental model. Nonetheless, the REKB, by capturing the three primary ingredients in the software creation exercise, goes much further towards accurately capturing this theory than approaches which rely purely on source code, bug reports, or document repositories. Once the theory has been (partially) captured, the problem of modifying a program solely by text production, as Naur refers to, can be omitted in favour of an approach which relies on solving the requirements problem.

The recent focus on adaptive requirements and adaptive software is, to my mind, misguided. This focus ignores four decades of evidence that we can *never* anticipate the direction our software will take, and disregards industry beliefs that incremental and iterative is the way to deliver high-business-value solutions. In fact, adaptive requirements demand even greater faith in up-front elicitation. Typically, adaptive software has focused on anticipating changes in tuneable parameters such as the number of servers or memory capacity. But it would be presumptuous of us to think we could anticipate not only the initial requirements (which conclusively is not possible) but also adaptive, high-level requirements. Such a focus flies in the face of Fred Brooks, who famously said to anticipate throwing the first version away. It also flies in the face of lean and agile software development, which place *human* decision-making at the centre of software development.

In this thesis I've made a contrarian argument. I've taken the position that instead we must be prepared, painful though it may be, to accept the inevitability of unanticipatability. If we cannot anticipate, we therefore must prepare to evolve when the situation demands. This means making formal our requirements, and keeping them accurate throughout the system's lifetime. It means changing the smallest necessary amount of code. It means making changes that are both timely and relevant.

My work has taken the stance that the REKB can be used in two ways. One is **verification**: what Zowghi and Gervasi [2003] call the proof obligation that must be discharged in building a system. A solution must show that the selected implementation S is complete with respect to the requirements G and domain assumptions D. The other is **search**: given a set of "possible" tasks in S, which of those will satisfy the requirements G? Zowghi and Gervasi make it clear they do not ascribe to the search approach: "this relation must not be regarded as a method to synthesize S and D". However, their reasoning is not presented. I suspect the hesitation is due to the difficulty in enumerating the possible solutions S, which after all are exponential even in the case where all the tasks are defined a priori, and infinite otherwise. But I am not suggesting that the abductive, search strategy is exhaustive. Rather, the intent is for the search to give some suggestions as to possible approaches. In the Zowghi and Gervasi approach, tasks in S are already present (although they are not present as such, we can call predicates added to S 'tasks'). In my approach, the tasks may be as optative as the goals themselves: suggestions that this task, *should it be implemented*, would satisfy the goal (using the subjunctive tense, to abuse Jackson [1997]'s grammar metaphor). This is the essence of the REKB approach.

Requirements problems should be seen as framing what is an essentially interactive and perpetually unfinished activity. Requirements are central to everything that an organization builds, the source of competitive advantage. Fred Brooks is quoted in Jarke et al. [2010] saying "[d]esign is not about solving fixed problems; it is constant framing of solution spaces". The typical approach to design only involves requirements at the very beginning of the design lifecycle. Abandoning requirements also abandons institutional knowledge, business rules, and design tradeoffs. Requirements should be maintained throughout the system's lifespan as the ultimate source of knowledge for evolving and updating systems to cope with what is the inevitability of change.

Bibliography

Carlos E Alchourrón, Peter Gärdenfors, and David Makinson. On the Logic of Theory Change: Partial Meet Contraction and Revision Functions. *Journal of Symbolic Logic*, 50(2):510–530, 1985.

Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. Location-Based Variability for Mobile Information Systems. In *International Conference on Advanced Information Systems Engineering*, volume 5074 of *Lecture Notes in Computer Science*, pages 575 – 578, Amsterdam, 2008. doi: 10.1007/978-3-540-69534-9.

Thomas A. Alspaugh, Stuart R. Faulk, Kathryn Heninger Britton, R. Alan Parker, David L. Parnas, and John E. Shore. Software Requirements for the A-7E Aircraft. Technical report, Naval Research Laboratory, 1992.

Scott W Ambler. Examining the "Big Requirements Up Front (BRUF)" Approach, July 2006.

David J Anderson. *Kanban*. Blue Hole Press, 2010.

Stuart Anderson and Massimo Felici. Controlling Requirements Evolution: An Avionics Case Study. In F Koornneef and M van Der Meulen, editors, *International Conference on Computer Safety, Reliability and Security*, volume LNCS 1943, page 0, Rotterdam, 2000.

Stuart Anderson and Massimo Felici. Requirements Evolution: From Process to Product Oriented Management. In *International Conference on Product Focused Software Process Improvement*, LNCS 2188, pages 27–41, Kaiserslautern, Germany, September 2001.

Annie I Antón. Goal-Based Requirements Analysis. In *International Conference on Requirements Engineering*, pages 136–144, Colorado Springs, Colorado, April 1996.

Annie I Antón and Colin Potts. Functional paleontology: system evolution as the user sees it. In *International Conference on Software Engineering*, pages 421–430, Toronto, Canada, 2001.

Grigoris Antoniou. The Role of Nonmonotonic Representations in Requirements Engineering. *International Journal of Software Engineering and Knowledge Engineering*, 8(3):385–399, 1998.

Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *International Conference on Software Engineering*, pages 298–308, Vancouver, September 2009. IEEE. doi: 10.1109/ICSE.2009.5070530.

Jorge Aranda, Steve M Easterbrook, and G Wilson. Requirements in the wild: How small companies do it. In *International Conference on Requirements Engineering*, Delhi, India, September 2007.

Albert Atkin. Peirce's Theory of Signs, October 2006.

Wade Baker, Alex Hutton, C David Hylender, Peter Lindstrom, and Denson Todd. Payment Card Industry Compliance Report. Technical report, Verizon, 2010.

Pierre F. Baldi, Cristina V. Lopes, Erik J. Linstead, and Sushil K. Bajracharya. A theory of aspects as latent topics. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 543–562, Nashville, 2008.

Luciano Baresi, Liliana Pasquale, and Paola Spoletini. Fuzzy Goals for Requirements-driven Adaptation. In *International Conference on Requirements Engineering*, Sydney, Australia, September 2010.

Victor Basili and Barry T Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27:42–52, 1984.

Bernardo Batiz-Lazo. Emergence and evolution of ATM networks in the UK, 1967-2000. *Business History*, 51(1):1–27, January 2009. doi: 10.1080/00076790802602164.

Roberto Battiti, Mauro Brunato, and Franco Mascia. *Reactive Search and Intelligent Optimization*, volume 45 of *Operations research/Computer Science Interfaces*. Springer Verlag, 2008. doi: 10.1007/978-0-387-09624-7.

L A Belady and Meir M Lehman. A model of large program development. *IBM Systems Journal*, 3: 225–252, 1976.

Nuel D. Belnap. A useful four-valued logic. In G Epstein and J M Dunn, editors, *Modern Uses of Multiple-Valued Logic*, pages 7–37. D. Reidel Publishing Co., Boston, 1977.

Eya Ben Charrada and Martin Glinz. An automated hint generation approach for supporting the evolution of requirements specifications. In *Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, page 58, Antwerp, Belgium, September 2010. doi: 10.1145/1862372.1862387.

Tim Berners-Lee and R Cailliau. WorldWideWeb: Proposal for a HyperText Project. Technical report, CERN, 1990.

Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *Trans. Comput. Syst.*, 2:39–59, 1984. doi: 10.1145/2080.357392.

David M. Blei, Andrew Y. Ng, and Michael I Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3(4-5):993–1022, May 2003. doi: 10.1162/jmlr.2003.3.4-5.993.

Jorgen Boegh. A New Standard for Quality Requirements. *IEEE Software*, 25(2):57–63, 2008. doi: 10.1109/MS.2008.30.

Barry Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5): 61–72, May 1988. doi: 10.1109/2.59.

Barry Boehm. Applying the Incremental Commitment Model to Brownfield System Development. In *Conference on Systems Engineering Research*, Loughborough, April 2009.

Barry Boehm, J R Brown, and M Lipow. Quantitative Evaluation of Software Quality. In *International Conference on Software Engineering*, pages 592–605, 1976.

B.W. Boehm and P.N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, 1988. doi: 10.1109/32.6191.

Juergen Boldt. The Common Object Request Broker: Architecture and Specification. Technical report, Object Management Group, July 1995.

David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture. Technical report, World Wide Web Consortium, February 2004.

Endre Boros and Peter L. Hammer. Pseudo-Boolean optimization. *Discrete Applied Mathematics*, 123 (1-3):155–225, November 2002. doi: 10.1016/S0166-218X(01)00341-9.

Craig Boutilier, Thomas Dean, and Steve Hanks. Planning under uncertainty: structural assumptions and computational leverage. In *New Directions in AI Planning*, pages 157–171. IOS Press, January 1996.

Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. Technical report, World Wide Web Consortium, May 2000.

Stewart Brand. *How Buildings Learn: What Happens After They're Built.* Viking Press, 1995.

Frederick P Brooks. *The mythical man-month.* Addison Wesley, Reading, Mass., 1st edition, 1975.

J A Bubenko. Information modeling in the context of system development. In *IFIP Congress*, pages 395–411, 1980.

E Capra, C Francalanci, and F Merlo. An Empirical Study on the Relationship Between Software Design Quality, Development Effort and Governance in Open Source Projects. *Trans. Soft. Eng.*, 2008.

Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: the Tropos project. *Information Systems*, 27:365–389, 2002.

Ned Chapin, Joanne E Hale, Juan Fernández-Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1): 3–30, January 2001. doi: 10.1002/smr.220.

David Chappell. The Trouble With CORBA. *Object News*, May 1998.

Marsha Chechik and Steve M Easterbrook. A framework for multi-valued reasoning over inconsistent viewpoints. In *International Conference on Software Engineering*, pages 411–420, Toronto, September 2001. IEEE Computer Society.

Betty H.C. Cheng, Daniel M Berry, and Ji Zhang. The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems. In *International Conference on Requirements Engineering: Foundation for Software Quality*, pages 113–120, Porto, Portugal, June 2005.

E Christensen, Francisco Curbera, G Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. Technical report, World Wide Web Consortium, March 2001.

Lawrence Chung, John Mylopoulos, and Brian A Nixon. Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. *Trans. Soft. Eng.*, 18:483–497, 1992.

Lawrence Chung, Brian A Nixon, and Eric S Yu. Dealing with change: An approach using non-functional requirements. *Requirements Engineering Journal*, 1(4):238–260, December 1996. doi: 10.1007/BF01587102.

Lawrence Chung, Brian A Nixon, Eric S Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*, volume 5 of *International Series in Software Engineering.* Kluwer Academic Publishers, Boston, October 1999.

Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature: a requirements engineering perspective. In *International Conference on Fundamental Approaches to Software Engineering*, pages 16–30, Budapest, April 2008.

Jane Cleland-Huang, C K Chang, and M Christensen. Event-based traceability for managing evolutionary change. *Trans. Soft. Eng.*, 29(9):796–810, September 2003. doi: 10.1109/TSE.2003.1232285.

Jane Cleland-Huang, Raffaella Settimi, Xuchang Zou, and Peter Solc. The Detection and Classification of Non-Functional Requirements with Application to Early Aspects. In *International Conference on Requirements Engineering*, pages 39–48, Minneapolis, Minnesota, 2006. IEEE. doi: 10.1109/RE.2006.65.

Francois Coallier. Software engineering – Product quality – Part 1: Quality model. Technical report, International Standards Organization - JTC 1/SC 7, 2001.

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

Stephen A. Cook. The complexity of theorem-proving procedures. In *Symposium on Theory of Computing*, pages 151–158, 1971. doi: 10.1145/800157.805047.

Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. An Architecture for Requirements-Driven Self-reconfiguration. In *International Conference on Advanced Information Systems Engineering*, pages 246–260, Amsterdam, 2009.

Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, April 1993. doi: 10.1016/0167-6423(93)90021-G.

A.S. d'Avila Garcez, A Russo, Bashar Nuseibeh, and Jeff Kramer. Combining abductive reasoning and inductive learning to evolve requirements specifications. *IEE Proceedings Software*, 150(1):25–38, 2003.

Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962. doi: 10.1145/368273.368557.

Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, March 1986. doi: 10.1016/0004-3702(86)90080-9.

Emanuele Di Rosa, Enrico Giunchiglia, and Marco Maratea. Solving satisfiability problems with preferences. *Constraints*, 15(4):485–515, July 2010. doi: 10.1007/s10601-010-9095-y.

Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien Nguyen. Refactoring-Aware Configuration Management for Object-Oriented Programs. In *International Conference on Software Engineering*, pages 427–436, Minneapolis, Minnesota, May 2007.

Joy Dixon and Jeffrey W Alexander. *The Thomson Nelson Guide to Writing in History*. Thomson Nelson, Scarborough, ON, 2006.

Jörg Doerr, D Kerkow, T Koenig, Thomas Olsson, and T Suzuki. Non-Functional Requirements in Industry - Three Case Studies Adopting an Experience-based NFR Method. In *International Conference on Requirements Engineering*, pages 373–384, 2005. doi: 10.1109/RE.2005.47.

W Dowling and Jean Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984. doi: 10.1016/0743-1066(84)90014-1.

Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, November 1979. doi: 10.1016/0004-3702(79)90008-0.

Jon Doyle. The ins and outs of reason maintenance. In *International Joint Conference on Artificial Intelligence*, pages 349–351, Karlsruhe, 1983. Department of Computer Science, Carnegie-Mellon University.

E Dubois, J Hagelstein, E Lahou, F Ponsaert, and A Rifaut. A knowledge representation language for requirements engineering. *Proceedings of the IEEE*, 74:1431–1444, 1986.

Pierre Duez and Kim J Vicente. Ecological interface design and computer network management: The effects of network size and fault frequency. *International Journal on Human-Computer Studies*, 63 (6):565–586, December 2005. doi: 10.1016/j.ijhcs.2005.05.001.

Juan J. Durillo, Yuanyuan Zhang, Enrique Alba, Mark Harman, and Antonio J. Nebro. A study of the bi-objective next release problem. *Empirical Software Engineering*, pages 1–32, December 2010. doi: 10.1007/s10664-010-9147-3.

Robert Dyer and Hridesh Rajan. Supporting dynamic aspect-oriented features. *ACM Transactions on Software Engineering and Methodology*, 20(2):1–34, August 2010. doi: 10.1145/1824760.1824764.

Steve M Easterbrook. Coordination breakdowns: why groupware is so difficult to design. In *Hawaii International Conference on System Sciences*, volume 4, pages 191–199, Honolulu, December 1995.

Steve M Easterbrook and Bashar Nuseibeh. Managing inconsistencies in an evolving specification. In *International Conference on Requirements Engineering*, pages 48–55, York, England, 1995.

Steve M Easterbrook and Bashar Nuseibeh. Using ViewPoints for Inconsistency Management. *IEE Software Engineering Journal*, 11(1):1–25, 1996.

Steve M Easterbrook, Eric S Yu, Jorge Aranda, Yuntian Fan, J Horkoff, Marcel Leica, and R A Qadir. Do viewpoints lead to better conceptual models? An exploratory case study. In *International Conference on Requirements Engineering*, pages 199–208, Paris, 2005.

Niklas Een, Alan Mischenko, and Nina Amla. A Single-Instance Incremental SAT Formulation of Proof- and Counterexample-Based Abstraction. In *Formal Methods in Computer-Aided Design (FMCAD)*, Lugano, Switzerland, October 2010.

Thomas Eiter and Georg Gottlob. The complexity of logic-based abduction. *Journal of the ACM*, 42 (1):3–42, January 1995. doi: 10.1145/200836.200838.

M Elvang-Gø ransson and Anthony Hunter. Argumentative logics: Reasoning with classically inconsistent information. *Data \& Knowledge Engineering*, 16(2):125–145, 1995. doi: 10.1016/0169-023X(95) 00013-I.

Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. Model evolution by run-time parameter adaptation. *International Conference on Software Engineering*, pages 111–121, 2009.

Neil A. Ernst and John Mylopoulos. Tracing software evolution history with design goals. In *International Workshop on Software Evolvability at ICSM*, Paris, France, October 2007.

Neil A. Ernst and John Mylopoulos. On the perception of software quality requirements during the project lifecycle. In *International Conference on Requirements Engineering: Foundation for Software Quality*, pages 143–157, Essen, Germany, June 2010.

Neil A. Ernst, John Mylopoulos, Yijun Yu, and Tien Nguyen. Requirements model evolution throughout the system lifecycle. In *International Conference on Requirements Engineering*, pages 321–322, Barcelona, September 2008. doi: 10.1109/RE.2008.11.

Neil A. Ernst, John Mylopoulos, Alex Borgida, and Ivan J Jureta. Reasoning with Optional and Preferred Requirements. In *International Conference on Conceptual Modeling (ER)*, pages 118–131, Vancouver, November 2010.

Neil A. Ernst, Alexander Borgida, and Ivan Jureta. Finding Incremental Solutions for Evolving Requirements. In *International Conference on Requirements Engineering*, Trento, Italy, September 2011.

T Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, June 2006. doi: 10.1016/j.patrec.2005.10.010.

Martin S. Feather and Steven Cornford. Quantitative risk-based requirements reasoning. *Requirements Engineering Journal*, 8:248–265, November 2003.

Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database.* MIT Press, 1998.

Susan Ferreira, Dan Shunk, James Collofello, Gerald Mackulak, and AmyLou Dueck. Reducing the risk of requirements volatility: findings from an empirical survey. *Journal of Software Maintenance and Evolution: Research and Practice*, pages n/a–n/a, October 2010. doi: 10.1002/smr.515.

Stephen Fickas and Martin S. Feather. Requirements monitoring in dynamic environments. In *International Conference on Requirements Engineering*, Washington, DC, USA, 1995.

Roy T Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California at Irvine, 2000.

R E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971. doi: 10.1016/0004-3702(71)90010-5.

Anthony Finkelstein and J. Dowell. A comedy of errors: the London Ambulance Service case study. In *International Workshop on Software Specification and Design*, pages 2–4, 1996. doi: 10.1109/IWSSD.1996.501141.

Anthony Finkelstein, M. Harman, S.A. Mansouri, Ren Jian, and Zhang Yuanyuan. 'Fairness Analysis' in Requirements Assignments. In *International Conference on Requirements Engineering*, pages 115–124, 2008. doi: 10.1109/RE.2008.61.

Anthony Finkelstein, Mark Harman, S. Mansouri, Jian Ren, and Yuanyuan Zhang. A search based approach to fairness analysis in requirement assignments to aid negotiation, mediation and decision making. *Requirements Engineering Journal*, 14(4):231–245, December 2009. doi: 10.1007/s00766-009-0075-y.

Kenneth D. Forbus and Johan de Kleer. *Building problem solvers.* MIT Press, Cambridge, MA, 1993.

Ariel Fuxman, Lin Liu, John Mylopoulos, Marco Roveri, and Paolo Traverso. Specifying and analyzing early requirements in Tropos. *Requirements Engineering Journal*, 9:132–150, 2004.

Harald Gall, Mehdi Jazayeri, René Klösch, and Georg Trausmuth. Software Evolution Observations Based on Product Release History. In *International Conference on Software Maintenance*, pages 160–166, Bari, Italy, 1997.

Artur Garcez, A Russo, Bashar Nuseibeh, and Jeff Kramer. An Analysis-Revision Cycle to Evolve Requirements Specifications. In *International Conference on Automated Software Engineering*, San Diego, USA, November 2001.

Peter Gärdenfors. Belief Revision : An Introduction. In Peter Gärdenfors, editor, *Belief Revision*, Cambridge Tracts in Theoretical Computer Science (No. 29), pages 1–28. Cambridge University Press, 1992.

Gregory Gay, Tim Menzies, Omid Jalali, Gregory Mundy, Beau Gilkerson, Martin Feather, and James Kiper. Finding robust solutions in requirements models. *Automated Software Engineering*, 17(1): 87–116, December 2009. doi: 10.1007/s10515-009-0059-7.

Daniel M. German. The GNOME project: a case study of open source, global software development. *Software Process: Improvement and Practice*, 8(4):201–215, 2003. doi: 10.1002/spip.189.

Daniel M. German, I Herraiz, Jesús M González-Barahona, and Gregorio Robles. On the prediction of the evolution of libre software projects. In *International Conference on Software Maintenance*, pages 405–414, Paris, October 2007.

A.K. Ghose. Formal tools for managing inconsistency and change in RE. In *International Workshop on Software Specification and Design*, pages 171–181, 2000. doi: 10.1109/IWSSD.2000.891138.

Paolo Giorgini, John Mylopoulos, Eleonora Nicchiarelli, and Roberto Sebastiani. Formal Reasoning Techniques for Goal Models. *Journal on Data Semantics*, 2800:1 − 20, 2003.

Tudor Girba and Stephane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18:207–236, 2006.

Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13(5), 1986.

Michael W. Godfrey, Abram Hindle, and Richard C. Holt. What's hot and what's not: Windowed developer topic analysis. In *International Conference on Software Maintenance*, pages 339–348, Edmonton, Alberta, Canada, September 2009. doi: 10.1109/ICSM.2009.5306310.

M.W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *Trans. Soft. Eng.*, 31(2):166–181, February 2005. doi: 10.1109/TSE.2005.28.

O. Gotel and Anthony Finkelstein. Contribution structures [Requirements artifacts]. In *International Conference on Requirements Engineering*, pages 100–107, September 1995. doi: 10.1109/ISRE.1995.512550.

Olly C Z Gotel and C W Finkelstein. An analysis of the requirements traceability problem. In *International Conference on Requirements Engineering*, pages 94–101, Colorado Springs, Colorado, 1994.

Sol Greenspan, John Mylopoulos, and Alex Borgida. Capturing more world knowledge in the requirements specification. In *International Conference on Software Engineering*, pages 225–234, Tokyo, 1982.

Richard Gronback. *Eclipse Modeling Project: A Domain-Specific Language Toolkit.* The Eclipse Series. Addison-Wesley Professional, 2009.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11(1):10–18, 2009.

S D P Harker, K D Eason, and J E Dobson. The change and evolution of requirements as a challenge to the practice of software engineering. In *International Conference on Requirements Engineering*, pages 266–272, 1993.

Mark Harman. Why Source Code Analysis and Manipulation Will Always be Important. In *Working Conference on Source Code Analysis and Manipulation*, pages 7–19, Timioara, Romania, September 2010. Ieee. doi: 10.1109/SCAM.2010.28.

Michi Henning. The Rise and Fall of CORBA. *ACM Queue*, 4, 2006.

Andrea Herrmann, Armin Wallnöfer, and Barbara Paech. Specifying Changes Only - A Case Study on Delta Requirements. In *International Conference on Requirements Engineering: Foundation for Software Quality*, pages 45 – 58, Amsterdam, 2009. doi: 10.1007/978-3-642-02050-6_5.

Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us? In *International Conference on Mining Software Repositories*, pages 99–108, Leipzig, May 2008. doi: 10.1145/1370750.1370773.

Abram Hindle, Neil A. Ernst, Michael W Godfrey, and John Mylopoulos. Automated topic naming to support cross-project analysis of software maintenance activities. In *International Conference on Mining Software Repositories*, Honolulu, May 2011.

Richard C. Holt, Abram Hindle, and Michael W Godfrey. Release Pattern Discovery via Partitioning: Methodology and Case Study. In *International Conference on Mining Software Repositories*, pages 19–27, Minneapolis, MN, May 2007. doi: 10.1109/MSR.2007.28.

Richard Hopkins and Kevin Jenkins. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. IBM Press, 2008.

Jennifer Horkoff and Eric Yu. Finding Solutions in Goal Models: An Interactive Backward Reasoning Approach. In *International Conference on Conceptual Modeling (ER)*, pages 59–75. Vancouver, November 2010. doi: 10.1007/978-3-642-16373-9_5.

Markus Horstmann and Mary Kirtland. DCOM Architecture. Technical report, Microsoft Corporation, Redmond, WA, 1997.

Anthony Hunter. Paraconsistent Logics. In D Gabbay and Ph Smets, editors, *Handbook of Defeasible Reasoning and Uncertain Information*. Kluwer, 1998.

IEEE Software Engineering Standards Committee. IEEE Recommended Practice for Software Requirements Specifications. Technical report, IEEE, 1998.

Paola Inverardi and Marco Mori. Feature oriented evolutions for context-aware adaptive systems. In *Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pages 93–97, Antwerp, Belgium, September 2010. doi: 10.1145/1862372.1862394.

Michael Jackson. The Meaning of Requirements. *Annals of Software Engineering*, 3:5–21, 1997.

Matthias Jarke, Pericles Loucopoulos, Kalle Lyytinen, John Mylopoulos, and William N. Robinson. The Brave New World of Design Requirements: Four Key Principles. In *International Conference on Advanced Information Systems Engineering*, pages 470–482, Hammaret, Tunisia, May 2010. doi: 10.1007/978-3-642-13094-6.

Brad Curtis Johnson. A Distributed Computing Environment Framework: An OSF Perspective. Technical report, The Open Software Foundation, June 1991.

Capers Jones. Software Quality in 2008 : A Survey of the State of the Art. Technical report, Software Quality Institute, 2008.

Ivan J Jureta, John Mylopoulos, and Stéphane Faulkner. Revisiting the Core Ontology and Problem in Requirements Engineering. In *International Conference on Requirements Engineering*, pages 71—-80, Barcelona, September 2008.

Ivan J Jureta, Alex Borgida, John Mylopoulos, Neil A. Ernst, Alexei Lapouchnian, and Sotirios Liaskos. Techne: A(nother) requirements modeling language. Technical report, Dept. Comput. Sci., University of Toronto, 2009a.

Ivan J Jureta, John Mylopoulos, and Stéphane Faulkner. A core ontology for requirements. *Applied Ontology*, 4(3-4):169–244, 2009b.

Ivan J Jureta, Alex Borgida, Neil A. Ernst, and John Mylopoulos. Techne: Towards a New Generation of Requirements Modeling Languages with Goals, Preferences, and Inconsistency Handling. In *International Conference on Requirements Engineering*, Sydney, Australia, September 2010.

Ivan J Jureta, Alexander Borgida, Neil A. Ernst, and John Mylopoulos. Guided Tour of Paraconsistent Formalisms for Language Designers in Requirements Engineering. *in preparation*, 2011.

Piotr Kaminski, Marin Litoiu, and Hausi Müller. A design technique for evolving web services. In Hakan Erdogmus, Eleni Stroulia, and Darlene A Stewart, editors, *Centre for Advanced Study Conference*, pages 303–317, 2006.

J Kephart and D Chess. The vision of autonomic computing. *IEEE Computer*, 36, 2003.

Günter Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. In Rachid Guerraoui, editor, *European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 351–366. Springer Berlin Heidelberg, November 1999. doi: 10.1007/3-540-48743-3.

Stefan Koch and Georg Schneider. Effort, co-operation and co-ordination in an open source software project: GNOME. *Information Systems Journal*, 12:27–42, 2002. doi: 10.1046/j.1365-2575.2002.00110.x.

Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference On Artificial Intelligence*, pages 1137–1143, Toronto, 1995.

Donald Kossmann, S. Borzsony, and K. Stocker. The Skyline operator. In *International Conference on Data Engineering*, pages 421–430, Heidelberg, 2001. doi: 10.1109/ICDE.2001.914855.

P. Krapivsky and S. Redner. Organization of growing random networks. *Physical Review E*, 63(6): 066123, May 2001. doi: 10.1103/PhysRevE.63.066123.

W Lam and Martin Loomes. Requirements Evolution in the Midst of Environmental Change: A Managed Approach. In *Euromicro Conference on Software Maintenance and Reengineering*, pages 121–127, Florence, Italy, March 1998. doi: 10.1109/CSMR.1998.665774.

Dean Leffingwell and Don Widrig. *Managing Software Requirements: A Use Case Approach*. Addison-Wesley Professional, 2nd edition, 2003.

Meir M Lehman and L A Belady. *Program Evolution: Processes of Software Change*. APIC Studies In Data Processing. Academic Press Professional, San Diego, 1985.

Meir M Lehman and Juan Fernández-Ramil. Software evolution. In N.H. Madhavji, J.C. Fernández-Ramil, and D.E. Perry, editors, *Software evolution and feedback: Theory and practice*, pages 7–40. Wiley, 2006.

Meir M Lehman and Juan F Ramil. Software evolution - Background, theory, practice. *Inf. Process. Lett.*, 88:33–44, 2003. doi: doi:10.1016/S0020-0190(03)00382-X.

Meir M Lehman, Juan F Ramil, P D Wernick, D E Perry, and W M Turski. Metrics and laws of software evolution-the nineties view. In *International Software Metrics Symposium*, pages 20–32, Albuquerque, NM, 1997.

Emmanuel Letier and Axel van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *SIGSOFT Conference on Foundations of Software Engineering*, pages 53—-62, Newport Beach, CA, 2004. ACM Press. doi: 10.1145/1029894.1029905.

H Levesque. Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23(2):155–212, July 1984. doi: 10.1016/0004-3702(84)90009-2.

Hector J. Levesque. *A formal treatment of incomplete knowledge bases*. Ph.d., University of Toronto, 1981.

Hector J. Levesque. A knowledge-level account of abduction. In *International Joint Conference on Artificial Intelligence*, pages 1061–1067, Detroit, August 1989.

Isaac Levi. Subjunctives, dispositions and chances. *Synthese*, 34(4):423–455, April 1977. doi: 10.1007/BF00485649.

Sotirios Liaskos, Alexei Lapouchnian, Yijun Yu, Eric S Yu, and John Mylopoulos. On Goal-based Variability Acquisition and Analysis. In *International Conference on Requirements Engineering*, Minneapolis, Minnesota, September 2006.

B. P. Lientz, E Burton Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466, 1978.

Marco Lormans. Monitoring Requirements Evolution using Views. In *Euromicro Conference on Software Maintenance and Reengineering*, pages 349–352, Amsterdam, March 2007.

Marco Lormans, H van Dijk, Arie van Deursen, E Nocker, and A de Zeeuw. Managing evolving requirements in an outsourcing context: an industrial experience report. In *Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pages 149–158, 2004.

Kalle Lyytinen and Mike Newman. Explaining information systems change: a punctuated socio-technical change model. *European Journal of Information Systems*, 17(6):589–613, December 2008. doi: 10.1057/ejis.2008.50.

Michael S Mahoney. Finding a History for Software Engineering. *IEEE Annals of the History of Computing*, 26:8–19, 2004. doi: 10.1109/MAHC.2004.1278847.

A. Marcus, A. Sergeyev, Václav T. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. In *International Working Conference on Reverse Engineering*, pages 214–223, November 2004. doi: 10.1109/WCRE.2004.10.

Bart Massey. Where Do Open Source Requirements Come From (And What Should We Do About It)? In *Workshop on Open source software engineering at ICSE*, Orlando, FL, USA, 2002.

Vlada Matena and Mark Hapner. Enterprise JavaBeansSpecification, v1.1. Technical report, Sun Microsystems, Palo Alto, CA, 1999.

J McCall. *Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisition Manager*, volume 1-3. General Electric, November 1977.

Christopher McDonald. 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):32–45, December 2010.

Qiaozhu Mei, Xuehua Shen, and ChengXiang Zhai. Automatic labeling of multinomial topic models. In *International conference on Knowledge discovery and data mining*, pages 490–499, San Jose, California, 2007. doi: 10.1145/1281192.1281246.

Tom Mens, Juan Fernández-Ramil, and S Degrandsart. The evolution of Eclipse. In *International Conference on Software Maintenance*, pages 386–395, Shanghai, China, October 2008.

Tim Menzies. On the Practicality of Abductive Validation. In *European Conference on Artificial Intelligence*, Budapest, 1996.

Tim Menzies, Steve M Easterbrook, Bashar Nuseibeh, and S Waugh. An empirical investigation of multiple viewpoint reasoning in requirements engineering. In *International Conference on Requirements Engineering*, pages 100–109, Limerick, Ireland, June 1999. doi: 10.1109/ISRE.1999.777990.

Tim Menzies, Steve M Easterbrook, B. A. Nuseibeh, and S. Waugh. Validating Inconsistent Requirements Models Using Graph-Based Abduction. Technical report, 2000.

Microsoft Corporation. DCOM Technical Overview. Technical report, Microsoft Corporation, Redmond, WA, November 1996.

Robert B. Miller. Response time in man-computer conversational transactions. In *AFIPS Fall Joint Computer Conference*, pages 267–277. ACM Press, December 1968. doi: 10.1145/1476589.1476628.

David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and Easy Distributions of SAT Problems. In *American Association for Artificial Intelligence*, pages 459–465, 1992.

A. Mockus and L.G. Votta. Identifying reasons for software changes using historic databases. In *International Conference on Software Maintenance*, pages 120–130, San Jose, CA, 2000. doi: 10.1109/ICSM.2000.883028.

John Mylopoulos, Alex Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing Knowledge About Information Systems. *IEEE Trans. Inf. Systems*, 8:325–362, 1990.

Vic Nanda and Nazim H Madhavji. The impact of environmental evolution on requirements changes. In *International Conference on Software Maintenance*, pages 452–461, Montreal, October 2002.

P Naur. Programming as theory building. *Microprocessing and Microprogramming*, 15(5):253–261, May 1985. doi: 10.1016/0165-6074(85)90032-8.

Israel Navarro, Nancy Leveson, and Kristina Lunqvist. Semantic decoupling : reducing the impact of requirement changes. *Requirements Engineering Journal*, 15:419–437, 2010. doi: 10.1007/s00766-010-0109-5.

Alan Newell. The Knowledge Level. *Journal of Artificial Intelligence*, 18, 1982.

Tien Nguyen, Ethan V Munson, John T Boyland, and Cheng Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *International Conference on Software Engineering*, pages 215–224, St. Louis, MI, May 2005. ACM Press. doi: 10.1145/1062455.1062504.

B. Nuseibeh. Ariane 5: Who Dunnit? *IEEE Software*, 14(3):15–16, May 1997. doi: 10.1109/MS.1997.589224.

B Nuseibeh, Steve M Easterbrook, and Alessandra Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 58(2):171–180, 2001. doi: 10.1016/S0164-1212(01)00036-X.

Ramon O'Callaghan. Fixing the payment system at Alvalade XXI: a case on IT project risk management. *Journal of Information Technology*, 22(4):399–409, December 2007. doi: 10.1057/palgrave.jit.2000116.

OSGi Alliance. OSGi Service Platform, Core Specification, Release 4. Technical report, OSGI, 2009.

David L Parnas. Designing software for ease of extension and contraction. In *International Conference on Software Engineering*, pages 264–277, Piscataway, NJ, USA, 1978.

David L Parnas. Software Aspects of Strategic Defense Systems. *Communications of the ACM*, 28:1326–1335, 1985.

PCI Security Standards Council. PCI DSS Requirements and Security Assessment Procedures, Version 2.0. Technical report, PCI, Boston, October 2010.

David Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–47, August 1988. doi: 10.1016/0004-3702(88)90077-X.

David Poole. A methodology for using a default and abductive reasoning system. *International Journal of Intelligent Systems*, 5(5):521–548, December 1990. doi: 10.1002/int.4550050506.

J Postel and J Reynolds. TELNET Protocol Specification. Technical report, Internet Engineering Task Force, 1983.

Nauman A. Qureshi, Anna Perini, Neil A. Ernst, and John Mylopoulos. Towards a Continuous Requirements Engineering Framework for Self-Adaptive Systems. In *Requirements at Run-time at RE*, Sydney, September 2010.

Nauman A Qureshi, Ivan J Jureta, and Anna Perini. Requirements Engineering for Self-Adaptive Systems : Core Ontology and Problem Statement. In *International Conference on Advanced Information Systems Engineering*, London, June 2011.

Steve Reeves and Mike Clarke. *Logic for computer science.* Addison Wesley, 2003.

Björn Regnell, Richard Berntsson Svensson, and Krzysztof Wnuk. Can We Beat the Complexity of Very Large-Scale Requirements Engineering? In *International Conference on Requirements Engineering: Foundation for Software Quality*, volume 5025 of *Lecture Notes in Computer Science*, pages 123–128, Montpellier, France, 2008. doi: 10.1007/978-3-540-69062-7.

H.B. Reubenstein and R.C. Waters. The Requirements Apprentice: automated assistance for requirements acquisition. *Trans. Soft. Eng.*, 17(3):226–240, March 1991. doi: 10.1109/32.75413.

Peter C. Rigby and Ahmed E Hassan. What Can OSS Mailing Lists Tell Us? A Preliminary Psychometric Text Analysis of the Apache Developer Mailing List. In *International Conference on Mining Software Repositories*, page 23, Leipzig, 2007.

William N. Robinson. Integrating multiple specifications using domain goals. In *International Workshop on Software Specification and Design*, pages 219–226, April 1989. doi: 10.1145/75199.75232.

William N. Robinson and S. Pawlowski. Surfacing root requirements interactions from inquiry cycle requirements documents. In *International Conference on Requirements Engineering*, pages 82–89, Colorado Springs, Colorado, April 1998. doi: 10.1109/ICRE.1998.667812.

William N. Robinson, Suzanne D. Pawlowski, and Vecheslav Volkov. Requirements interaction management. *ACM Computing Surveys*, 35(2):132, 2003.

Douglas T Ross and Kenneth E Schoman. Structured Analysis for Requirements Definition. *Trans. Soft. Eng.*, 3(1):6–15, 1977.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A modern approach.* Prentice-Hall, New Jersey, 1995.

Alessandra Russo and Bashar Nuseibeh. Using Abduction to Evolve Inconsistent Requirements Specifications. *Australasian Journal of Information Systems*, 6(2):118–130, 1999.

Mehrdad Sabetzadeh, Shiva Nejati, Sotirios Liaskos, Steve M Easterbrook, and Marsha Chechik. Consistency Checking of Conceptual Models via Model Merging. In *International Conference on Requirements Engineering*, New Delhi, India, October 2007.

Rick Salay, John Mylopoulos, and Steve M Easterbrook. Using Macromodels to Manage Collections of Related Models. In Pascal Eck, Jaap Gordijn, and Roel Wieringa, editors, *International Conference on Advanced Information Systems Engineering*, volume 5565 of *Lecture Notes in Computer Science*, pages 141–155–155, Hammaret, Tunisia, 2009. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-02144-2.

Moshood Omolade Saliu and Guenther Ruhe. Bi-objective release planning for evolving software systems. In *SIGSOFT Conference on Foundations of Software Engineering*, page 105, New York, New York, USA, September 2007. ACM Press. doi: 10.1145/1287624.1287641.

Pete Sawyer, Nelly Bencomo, Jon Whittle, Emmanuel Letier, and Anthony Finkelstein. Requirements-Aware Systems. In *International Conference on Requirements Engineering*, Sydney, 2010.

Walt Scacchi. Understanding the requirements for developing open source softwaresystems. *IET Software*, 149(1):24–39, 2002. doi: 10.1049/ip-sen:20020202.

Walt Scacchi, Chris Jensen, John Noll, and Margaret Elliott. Multi-Modal Modeling, Analysis and Validation of Open Source Software Requirements Processes. In *International Conference on Open Source Software*, volume 1, pages 1–8, Genoa, Italy, July 2005.

John R. Searle. A Classification of Illocutionary Acts. *Language in Society*, 5(1):1 – 23, 1976.

Roberto Sebastiani, Paolo Giorgini, and John Mylopoulos. Simple and Minimum-Cost Satisfiability for Goal Models. In *International Conference on Advanced Information Systems Engineering*, pages 20–35, Riga, Latvia, June 2004.

Bart Selman and Hector J Levesque. Abductive and Default Reasoning: A Computational Core. In *American Association for Artificial Intelligence*, pages 343–348, Boston, August 1990.

Bart Selman, Hector Levesque, and David Mitchell. A New Method for Solving Hard Satisfiability Problems. In *American Association for Artificial Intelligence*, pages 440–446, San Jose, California, July 1992.

Bart Selman, Henry Kautz, and Bram Cohen. Local Search Strategies for Satisfiability Testing. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, October 1993.

Mary Shaw. Prospects for an Engineering Discipline of Software. *IEEE Software*, 7(6):15–24, 1990. doi: 10.1109/52.60586.

Vítor E. Silva Souza, Alexei Lapouchnian, William N. Robinson, and John Mylopoulos. Awareness Requirements for Adaptive Systems. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, Honolulu, September 2011.

Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide*. Wiley, New York, NY, USA, April 1997.

Jonathan B Spira. 20 Years-One Standard: The Story of TCP/IP. *Iterations: An Inter-disciplinary Journal of Software History*, 2:1–3, April 2003.

W Edward Steinmueller. The U.S. software industry : an analysis and interpretative history. Research memoranda, Maastricht Economic Research Institute on Innovation and Technology, Maastricht, 1995.

Sun Microsystems Inc. RPC: Remote Procedure Call. Technical report, Internet Engineering Task Force, April 1988.

Sun Microsystems Inc. Java Remote Method Invocation. Technical report, Sun Microsystems, Palo Alto, CA, 1999.

E Burton Swanson. The dimensions of maintenance. In *International Conference on Software Engineering*, pages 492–497, San Francisco, California, 1976.

Daisuke Tanabe, Kohei Uno, Kinji Akemine, Takashi Yoshikawa, Haruhiko Kaiya, and Motoshi Saeki. Supporting Requirements Change Management in Goal Oriented Analysis. In *International Conference on Requirements Engineering*, pages 3–12, Barcelona, 2008. doi: 10.1109/RE.2008.18.

H. Thimbleby. Delaying commitment. *IEEE Software*, 5(3):78–86, 1988. doi: 10.1109/52.2027.

Christoph Treude and Margaret-Anne Storey. ConcernLines: A timeline view of co-occurring concerns. In *International Conference on Software Engineering*, pages 575–578, Vancouver, May 2009.

Qiang Tu and M W Godfrey. An integrated approach for studying architectural evolution. In *International Conference on Program Comprehension*, pages 127–136, 2002.

Thein Than Tun, Yijun Yu, Robin Laney, and Bashar Nuseibeh. Recovering Problem Structures to Support the Evolution of Software Systems. Technical report, Open University, Milton Keynes, 2008.

Axel van Lamsweerde. Goal-oriented requirements enginering: a roundtrip from research to practice. In *International Conference on Requirements Engineering*, pages 4–7, 2004.

Axel van Lamsweerde. Requirements engineering: from craft to discipline. In *SIGSOFT Conference on Foundations of Software Engineering*, pages 238–249, Atlanta, Georgia, November 2008.

Axel van Lamsweerde. Reasoning About Alternative Requirements Options. In Alexander Borgida, Vinay K Chaudhri, Paolo Giorgini, and Eric S K Yu, editors, *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 380–397. Springer, 2009.

Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *Trans. Soft. Eng.*, 26:978–1005, 2000.

Axel van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089–1114, 1998. doi: 10.1109/32.738341.

Axel van Lamsweerde, Emmanuel Letier, and Robert Darimont. Managing Conflicts in Goal-Driven Requirements Engineering. *Trans. Soft. Eng.*, 24(11):908–926, November 1998. doi: 10.1109/32.730542.

Walter G Vincenti. *What engineers know and how they know it : analytical studies from aeronautical history*. Johns Hopkins University Press, Baltimore, 1993.

Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 35, February 1997.

Steve Vinoski. REST Eye for the SOA Guy. *IEEE Internet Computing*, 11:82–84, 2007.

Yiqiao Wang and John Mylopoulos. Self-Repair through Reconfiguration: A Requirements Engineering Approach. In *International Conference on Automated Software Engineering*, pages 257–268. IEEE, November 2009. doi: 10.1109/ASE.2009.66.

Yiqiao Wang, Sheila A. McIlraith, Yijun Yu, and John Mylopoulos. Monitoring and diagnosing software requirements. *Automated Software Engineering*, 16:3–35, 2009. doi: 10.1007/s10515-008-0042-8.

P D Wernick, T Hall, and Chrystopher Nehaniv. Software Evolutionary Dynamics Modelled as the Activity of an Actor-Network. In *International Workshop on Software Evolvability at ICSM*, pages 74–81, 2006.

David Wheeler. SLOCcount, 2009.

J E White. A High-Level Framework for Network-Based Resource Sharing. Technical report, Internet Engineering Task Force, March 1975.

Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H.C. Cheng, and Jean-Michel Bruel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *International Conference on Requirements Engineering*, pages 79–88, Atlanta, August 2009. doi: 10.1109/RE.2009.36.

Karl E. Wiegers. *Software Requirements*. Microsoft Press, 2003.

Roel Wieringa, Neil Maiden, Nancy Mead, and Colette Rolland. Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requirements Engineering Journal*, 11: 102–107, March 2006.

Dave Winer. XML/RPC Specification. Technical report, Userland Software, 1999.

Krzysztof Wnuk, Björn Regnell, and Lena Karlsson. What Happened to Our Features? Visualization and Understanding of Scope Change Dynamics in a Large-Scale Industrial Setting. In *International Conference on Requirements Engineering*, pages 89–98. IEEE, August 2009. doi: 10.1109/RE.2009.32.

C. Wohlin and A. Aurum. What is important when deciding to include a software requirement in a project or release? In *International Symposium on Empirical Software Engineering*, pages 237–246, 2005. doi: 10.1109/ISESE.2005.1541833.

Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. Understanding Feature Evolution in a Family of Product Variants. In *International Working Conference on Reverse Engineering*, pages 109–118. IEEE, October 2010. doi: 10.1109/WCRE.2010.20.

Yijun Yu, John Mylopoulos, and J Sampai. From goals to aspects: discovering aspects from requirements goal models. In *International Conference on Requirements Engineering*, pages 33–42, Kyoto, Japan, September 2004.

Yijun Yu, Julio Cesar Sampaio Do Prado Leite, Lin Liu, Eric S Yu, and John Mylopoulos. Quality-based Software Reuse. In *International Conference on Advanced Information Systems Engineering*, pages 535–550, Porto, Portugal, June 2005a.

Yijun Yu, Yiqiao Wang, John Mylopoulos, Sotirios Liaskos, and Alexei Lapouchnian. Reverse engineering goal models from legacy code. In *International Conference on Requirements Engineering*, pages 363–372, 2005b.

Pamela Zave and Michael Jackson. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6:1–30, 1997.

Yuanyuan Zhang, Mark Harman, and S. Afshin Mansouri. The multi-objective next release problem. In *Genetic And Evolutionary Computation Conference*, pages 1129 – 1137, London, 2007.

Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. Search Based Requirements Optimisation: Existing Work & Challenges. In *International Conference on Requirements Engineering: Foundation for Software Quality*, volume 5025 of *Lecture Notes in Computer Science*, pages 88–94, Montpellier, France, 2008. doi: 10.1007/978-3-540-69062-7.

Didar Zowghi and Vincenzo Gervasi. On the interplay between consistency, completeness, and correctness in requirements evolution. *Information and Software Technology*, 45(14):993–1009, November 2003. doi: 10.1016/S0950-5849(03)00100-9.

Didar Zowghi and R Offen. A logical framework for modeling and reasoning about the evolution of requirements. In *International Conference on Requirements Engineering*, pages 247–257, 1997.