

# Requirements evolution drives software evolution

Neil A. Ernst  
Dept. of Computer Science  
University of Toronto  
nernst@cs.toronto.edu

Alexander Borgida  
Dept. of Computer Science  
Rutgers University  
borgida@cs.rutgers.edu

John Mylopoulos  
Dept. of Inf. Eng. and  
Computer Science  
University of Trento  
jm@disi.unitn.it

## ABSTRACT

Changes to software should be made with reference to the requirements of that software, as these requirements provide the reasons for a change. Requirements serve to tie the implementation world of the developers to the problem world of the stakeholders. Most empirical studies of requirements have shown that misunderstood and changing requirements cause the majority of failures and costs in software. However, research in software evolution has typically focused on *how* to evolve software and not *why*. In our view, evolving software is about solving requirements problems, that is, finding new implementations which will satisfy the requirements while respecting domain assumptions. We argue that by describing this relationship, an implementation choice that best meets stakeholder needs can be made. We describe a tool that models requirements problems. This tool can find incremental solutions to evolving requirements problems quickly.

## Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications

## General Terms

Management

## Keywords

Requirements evolution; unanticipated change; goal-oriented modeling

## 1. INTRODUCTION

Software is developed, maintained, and evolved in order to satisfy stakeholder requirements. Requirements, whether formal or informal, implicit or explicit, serve to define the nature and quality of the purpose of a software machine. Defining a machine's purpose this way is the *requirements problem*: to elicit the desired characteristics of a machine

(specification  $S$ ) that will bring about some desired properties (the requirements  $R$ ) in an environment or world  $W$ . This was formalized by Zave and Jackson [24] as satisfying the condition:

$$W, S \vdash R$$

where  $W \cup S$  is consistent, and  $\vdash$  is some form of logical deduction relationship.

We argue that without reference to the requirements problem, the reasons for evolving software are lost, and without these reasons, we cannot be sure that a) we are addressing the stakeholder's requirements and b) that our choices are optimal.

How does our view of the importance of the system's requirements fit with the literature on software evolution, which is primarily focused on implementation-level artifacts such as source code? This issue has long been identified, if not explicitly addressed by research initiatives. For example, a 2009 presentation [14] listed several key challenges for software evolution. One challenge for software migration was "How to ensure that the resulting system has the desired quality and functionality?" This question is the motivation for our work in requirements evolution. Requirements give the answer to the questions of what the desired quality is (typically via non-functional requirements such as usability). Requirements also define the stakeholder acceptance criteria for functionality. This motivates our (deliberately provocative) title, that requirements evolution drives software evolution. Another publication [15] raised the challenge of accommodating "... evolution of higher-level artifacts such as analysis and design models, software architectures, requirement specifications, and so on." Focusing on higher levels of abstraction is not merely useful but essential. While corrective software maintenance may be performed without reference to earlier phases of development, adaptive maintenance must reflect the stakeholder goals captured by requirements.

We can identify the importance of requirements in adaptive maintenance in at least two cases. For one, in many projects a common way to elicit requirements is requirements re-use, which is associated with lower requirements volatility [7]. A second case is the merge of two similar systems, as might occur in a corporate takeover. The central question is what combination of capabilities is meeting the requirements, and which are superfluous to needs [12].

There seem to be two objections to making requirements more prominent in the study of software evolution. (1) For many people, source code is the only accurate artifact [8]. Requirements are either not used explicitly, or are stale the moment they are 'completed'. (2) In terms of quantity, most

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IWPSE-EVOL'11, September 5–6, 2011, Szeged, Hungary.  
Copyright 2011 ACM 978-1-4503-0848-9/11/09 ...\$10.00.

change tasks involve low-level corrective maintenance tasks, rather than more complex adaptive maintenance. Therefore, the objection goes, efforts are better focused there.

The problem of requirements relevance (1) 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. 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.

As for objection (2), while the numbers of corrective change tasks might seem greater, our 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.

Our proposition is to use requirements documents as drivers for implementation decisions. Such documents force the requirements to be considered explicitly. These documents might take the form of lengthy lists, as most industrial tools currently do; preferably, and in our implementation, they would be well-structured graphs that represent all aspects of the requirements problem, capturing stakeholder objectives, domain assumptions, and implementation options. In our view, such models can be used to capture changes in the problem, and the key challenge is ‘requirements repair’: re-evaluating the available solutions to solve the changed requirements, adding (minimal) new implementations where it is necessary [16].

We begin by outlining other work which supports our viewpoint. Following that, we introduce a brief case study we conducted, then describe a framework for unifying requirements, domain constraints, and implementation. The framework considers software development an activity that solves a requirements problem. The main contribution of this paper is to show how this framework can be used in the case of problem evolution to find optimal solutions. We describe important operators for describing these solutions, using examples from our case study. We then present our tool and some of our work using it to solve requirements evolution problems in practice.

## 2. RELATED WORK

We are not the only researchers to make the observation that requirements are vitally important artifacts in software evolution.

The seminal work on software evolution is that of Les Belady and Manny Lehman. 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.” [2, p. 239]. Swanson, in [22], similarly identifies “changes in data and processing environments” as a major cause of adaptive maintenance activity. Most software engineering texts (e.g., [21]) discuss the need to deal with changes in requirements, although they commonly focus on changes pre-delivery. One way of doing this is to co-evolve requirements and tests to preserve requirements relevance [3].

More recently, there has been a great deal of research looking at run-time requirements driven adaptation. With the RELAX framework [23], 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 [1]. Research has also looked at using control variables in conjunction with awareness requirements to dynamically adjust to new contexts [20].

Unlike in adaptation, where changes must be anticipated, our primary focus is on *unanticipated* requirements change. [10] describe a system which deals with unanticipated changes by asking users what steps to take next. Our work differs in treating changes at the requirements level. [9] identifies the need for “delta requirements”, requirements which must be added subsequent to software delivery. Similarly, [17] points out the importance of environmental evolution on the requirements of an existing system.

## 3. CASE STUDY: PAYMENT CARD SECURITY REQUIREMENTS

To explore the issues involved in requirements evolution we conducted a case study to highlight how requirements changes impact software systems.

The Payment Card Industry Security Standards Council<sup>1</sup> 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 Data Security Standard (henceforth PCI-DSS) [19] specifically applies to transactions between merchant and payment processor.

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 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.

The 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 goals:

1. Build and Maintain a Secure Network
2. Protect Cardholder Data
3. Maintain a Vulnerability Management Program
4. Implement Strong Access Control Measures
5. Regularly Monitor and Test Networks
6. 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 our modeling language, these are alternative solutions.

<sup>1</sup><https://www.pcisecuritystandards.org>

To study this standard in use, we derived requirements for operating a football stadium from another case study [18], 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.

### 3.1 Requirements Evolution in PCI-DSS

Our examination of this case study revealed some important scenarios where changing requirements drove an adaptive change in implementation. We classify these scenarios into three categories using the language of belief revision.

First, we have the case where requirements have been *expanded*. An example of this is the added requirement, in PCI-DSS version 2, to rank system vulnerabilities explicitly by 2012. To be compliant, some implementation task must be carried out to perform this ranking (e.g., a system scanner that can automatically detect vulnerabilities).

Secondly, we have *contraction* of the requirements. Where before one could use WEP (Wired Equivalent Privacy) to encrypt wireless transmissions, the new version of the standard removes this option, leaving only WPA (WiFi Protected Access). This implies that our implementation of wireless security (now inconsistent with the requirements) must be resolved to remove the conflict.

Finally, we can *revise* requirements. In the PCI-DSS, such a scenario occurred when the demand to use “cryptography” was revised to reflect that “strong cryptography” must be used. Our revised implementation must add this new requirement and remove possible inconsistent assumptions about what ‘strong’ might mean. This might conflict with our choice of cryptography protocols. On the other hand, we might have used strong encryption in one place but not others, in which case we can re-use that solution.

## 4. THE REQUIREMENTS PROBLEM

As we discussed in the introduction, the *requirements problem* can be formally defined as given  $R$ , find  $W$  and  $S$  such that  $W, S \vdash R$  ( $W$  being the underlying context or world,  $S$  the specification, and  $R$  the requirements). Note that a final, running system, achieving the requirements  $R$ , is obtained when the specifications  $S$  are actually implemented.

This characterization is overly simplified. First, given  $W$  and  $R$ , there can be *many* solutions  $S$  to the requirements problem. Therefore, provisions should be made for exploring the space of solutions and comparing them. 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”. The requirements problem is therefore two-staged: first, find solutions that satisfy mandatory requirements; second, decide on an optimal solution with respect to stakeholder priorities.

### 4.1 Techne and the Requirements Problem

To address the requirements problem, in [11] we introduced a new requirements modeling language, called *Techne*. *Techne* models requirements problems as composed of *requirements*, that is, desired properties; *tasks*, which are implementation actions to carry out to achieve the requirements; and *domain assumptions*, constraints and properties of the world. We further decompose requirements into softgoals, which have no clear criteria for being achieved; goals, stakeholder objectives; and quality constraints, which define a

quality space for softgoal satisfaction. We define several relationships which act on these components. *Refinements* are defined as modus ponens inferences, that is, inference applications from a set of antecedents to a single consequent (making our language propositional Horn logic). *Conflicts* are binary relationships between requirements  $x$  and  $y$ , allowing us to state that a certain solution cannot consist of both  $x$  and  $y$ . The unary relation *optionality* is used to indicate whether a particular element is required or preferred, and *preference* relations are binary relationships on any two components, indicating that the former is strictly preferred to the latter. Variation points exist wherever there exist multiple refinements of a requirement. Variation points define solution variability, so that one *Techne* model expresses zero or more potential solutions. In selecting a particular solution (that is, a set of tasks), we collapse the variability space. We then mark these tasks as “implemented” and wait for possible changes to the model.

In the payment card case, we have several sets of solutions. Variability comes from implementation decisions, as well as business focus areas. As we refine our requirements, we find fewer possible variations until we get to leaf level tasks: for example, the PCI-DSS security requirements are quite invariant, since all tasks must be carried out (you cannot ‘choose’ to carry out only a portion of the PCI-DSS requirements).

PCI-DSS is well-suited to representation in *Techne*. We map requirements as goals, and constraints as domain assumptions (for example, the football team must use legacy IBM devices). Tasks represent conformance tests in the PCI-DSS, or committed actions in the domain-specific model. We make further domain assumptions about the refinement of goals into (eventual) tasks. Example 1 illustrates this.

EXAMPLE 1. *Our elicitation with the stakeholders for the football stadium has produced the following set of communications sorted into tasks  $T$ , goals  $G$ , and domain assumptions  $D$ . A subset of the model, augmented with justifications connecting goals, tasks, etc.:*

- $G_{IncRev}$ : Increase revenue. (A mandatory goal.)
- $G_{AccCC}$ : Accept payment cards. (A mandatory goal.)
- $G_{NoLoss}$ : Avoid financial losses and penalties.
- $G_{PCIcompl}$ : Be PCI-DSS compliant.
- $D_1$ : The payment system will have a peak load of 70 concurrent users.
- $D_2$ : The stadium management do not have resources to purchase multiple servers.
- $T_1$ : Ensure virtualization instances are isolated from one another.
- $G_{2.2.1}$ : Implement only one primary function per server.
- $I_2$ : Satisfying  $G_{PCIcompl}$  satisfies  $G_{NoLoss}$ .
- $I_3$ : Satisfying  $G_{NoLoss}$  satisfies  $G_{IncRev}$ .
- $C_1$ :  $G_{2.2.1}$  and  $D_2$  are in conflict.

### 4.2 Finding Solutions to Static Problems

We first consider the case of finding solutions to requirements problems which are not changing. In [5] we provided two knowledge-level operations which can be used to do this. Our first operation, ARE\_GOALS\_ACHIEVED\_FROM,

concerns exploration from the implementation level. For example, assume we have an existing set of tasks implemented (common in maintenance situations). In this case, we would like to know whether the higher-level requirements (goals) can be satisfied by what we already have. This is known as bottom-up analysis in requirements modeling. Our new insight is that in evolving models, this operation takes on new significance, since it supports the question, “Must we do anything” to comply with new requirements. For example, if in our scenario no wireless access points exist, and there is a new PCI-DSS requirement that only WPA authentication is acceptable, the answer will be (happily) “No”.

ARE\_GOALS\_ACHIEVED\_FROM takes a requirements problem formulation, a set of tasks assumed to be implemented, and a set of goals to satisfy. It returns TRUE if the implementation can satisfy the goals. By itself, it does not solve the requirements problem, but on the other hand it has a much lower computational complexity than our second operation (linear-time for propositional Horn clauses).

Our second operation computes minimal sets of tasks which solve the given requirements problem. It provides *abductive explanations* of how the goals can be achieved from the tasks [13]. We call this operation MINIMAL\_GOAL\_ACHIEVEMENT. It takes as arguments a requirements problem formulation and a set of goals to check. It returns minimal sets of tasks which can satisfy those goals. Since it uses abductive inference, its complexity is NP-hard, but we have devised techniques for approximating this. For example, we would like to know what tests must be performed to satisfy  $G_{IncRev}$ , increase revenues. Our operation will return sets of minimal sets of tasks to be carried out, amongst which the stakeholder can choose, or further optimization identified (such as least cost).

### 4.3 Finding Solutions for Evolving Problems

The more interesting situation is when the requirements problem has changed. Given possible changes in goals, domain assumptions, and tasks, the requirements problem can be restated as finding new specifications which satisfy some property  $\Pi$  with respect to the original solution.

We define a function, GET\_MIN\_CHANGE\_TASK\_ENTAILING, which can find answers to this problem statement. The operation takes a requirements problem formulation, a set of implementation tasks which form the “original” solution, and a property  $\Pi$ . It returns sets of sets of tasks which satisfy the goals, and are minimal with respect to the distance function chosen. In our example, we came across the scenario where one supplier’s solution had failed – a proprietary mobile point of sale system [18]. In choosing the new supplier, clearly a key concern is how to re-use the extensive initial investment (such as back-end server technology, database licences, etc.). GET\_MIN\_CHANGE\_TASK\_ENTAILING can determine which re-use solution is most advantageous.

### 4.4 Selecting New Solutions

What properties  $\Pi$  should we consider in our problem? We look at four possibilities:

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. This is also known as “not-invented-here” syndrome.
2. *Minimal change effort solutions:* These approaches look

for solutions that minimize the extra effort required to implement the new specification. In our view of solutions as sets of tasks, this may be taken as “set subtraction”, in which case one could look for (i) the smallest difference cardinality or (ii) smallest difference cardinality *and* least size.

3. *Maximal familiarity solutions:* These approaches look for solutions that maximize the set of tasks used in the current solution, while implementing the new solution. One might prefer such an approach because it preserves most of the structure of the current solution.
4. *Solution reuse over history of changes:* Since the software has probably undergone a series of changes, each resulting in newly implemented task sets, one can try to maximize reuse of these (and thereby even further minimize current extra effort). This might be the case in software product lines, or in long-lived projects.

The above make it clear that there is unlikely to be a single optimal answer to the problem of requirements evolution. The best approach is to support developers and stakeholders in exploring alternatives.

## 4.5 Inconsistencies

In our treatment of the requirements evolution problem, we must deal with the possibility of inconsistency, since changes to a model often introduce incompatibilities. Inconsistency can only arise if introduced as a domain assumption that states the incompatibility of two other elements of the model; i.e., for goals  $G_1, G_2$ , we assert  $D_a : \{G_1 \wedge G_2 \rightarrow \perp\}$ . In the case of such an assertion, if the goals  $G_1$  and  $G_2$  can be satisfied through the assertion of task satisfaction, we would normally have an inconsistent theory and everything could be concluded from it. Our system’s reasoning is paraconsistent in the sense that only those solutions (sets of tasks  $S$ ) are proposed which satisfy the condition that not both  $G_1$  and  $G_2$  can be satisfied from  $S$ .

## 5. IMPLEMENTATION AND EVALUATION

We implemented our three operations using a combination of custom problem solver and assumption-based truth maintenance system (ATMS, [4]). Our problem solver understands models of requirements problems using our knowledge-level operations. When the user introduces new symbols and assertions, the problem solver stores these in the ATMS, which supports incremental and paraconsistent reasoning. The ATMS maintains the (minimal) explanations for why a particular goal is satisfied. It is incremental in that new assertions do not cause a complete re-calculation of these explanations. We leverage this incrementality for computing answers to our GET\_MIN\_CHANGE\_TASK\_ENTAILING operation. The problem solver interface contains techniques to reduce the scope of the problem. It also enhances the ATMS to support assertion retraction.

We have made two major approaches to validating our ideas. Our first approach was to verify the phenomenon of requirements change. In [6], we investigated the change of non-functional requirements in various open-source projects. Using machine learning techniques, we showed that the importance of NFRs was not correlated to time, but instead caused by external events (such as usability reviews). Following this, we evaluated our techniques for incremental re-

vision using simulation and our case study. We generated random models and showed that our incremental solution finding performed dramatically faster than other approaches [5]. Finally, we have applied our reasoner to the PCI-DSS case study. The reasoner has derived a list of new acceptance tests which the stakeholder must pass to remain compliant to the standard.

## 6. CONCLUSIONS AND FUTURE WORK

In this paper we argued that without consideration of the requirements, choosing which change activity to undertake is difficult. Requirements are the bridge between the problem domain and the implementation domain. Consequently, they act as the interface between the stakeholder perspective and the developer perspective. When we choose a change task, we should choose one which best satisfies the revised requirements problem, because this will maximize stakeholder satisfaction. We introduced some operations which can help with this decision, and a tool we implemented to allow for decision support.

There are several avenues to explore. We need to further investigate the traceability/co-evolution challenge listed in [15], because requirements are not typically maintained throughout the software lifecycle. We are also interested in qualitative studies of requirements evolution in practice.

## 7. REFERENCES

- [1] L. Baresi, L. Pasquale, and P. Spoletini. Fuzzy Goals for Requirements-driven Adaptation. In *Int. Conf. on Req. Engineering*, Sydney, Australia, Sept. 2010.
- [2] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 3:225–252, 1976.
- [3] E. Ben Charrada and M. 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, Sept. 2010.
- [4] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, Mar. 1986.
- [5] N. A. Ernst, A. Borgida, and I. Jureta. Finding Incremental Solutions for Evolving Requirements. In *Int. Conf. on Req. Engineering*, Trento, Italy, Sept. 2011.
- [6] N. A. Ernst and J. 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.
- [7] S. Ferreira, D. Shunk, J. Collofello, G. Mackulak, and A. 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, Oct. 2010.
- [8] M. Harman. Why Source Code Analysis and Manipulation Will Always be Important. In *Working Conference on Source Code Analysis and Manipulation*, pages 7–19, Timișoara, Romania, Sept. 2010.
- [9] A. Herrmann, A. Wallnöfer, and B. 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.
- [10] P. Inverardi and M. 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, Sept. 2010.
- [11] I. J. Jureta, A. Borgida, N. A. Ernst, and J. Mylopoulos. Techne: Towards a New Generation of Requirements Modeling Languages with Goals, Preferences, and Inconsistency Handling. In *Int. Conf. on Req. Engineering*, Sydney, Australia, Sept. 2010.
- [12] R. Land and I. Crnkovic. Oh Dear, We Bought Our Competitor: Integrating Similar Software Systems. *IEEE Software*, 28(March/April):75–82, 2011.
- [13] H. J. Levesque. A knowledge-level account of abduction. In *International Joint Conference on Artificial Intelligence*, pages 1061–1067, Detroit, Aug. 1989.
- [14] T. Mens. Future Research Challenges in Software Evolution. In *Presentation to ERCIM Working Group on Software Evolution*, Brussels, 2009.
- [15] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in Software Evolution. In *Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pages 13–22, Lisbon, Sept. 2005.
- [16] J. Mylopoulos. The Requirements Problem Revisited. In *Presentation to IFIP Working Group 2.9*, Cancún, Feb. 2011.
- [17] V. Nanda and N. H. Madhavji. The impact of environmental evolution on requirements changes. In *International Conference on Software Maintenance*, pages 452–461, Montreal, Oct. 2002.
- [18] R. O’Callaghan. Fixing the payment system at Alvalade XXI: a case on IT project risk management. *Journal of Information Technology*, 22(4):399–409, Dec. 2007.
- [19] PCI Security Standards Council. PCI DSS Requirements and Security Assessment Procedures, Version 2.0. Technical report, PCI, Boston, Oct. 2010.
- [20] V. E. Silva Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos. Awareness Requirements for Adaptive Systems. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, Honolulu, Sept. 2011.
- [21] I. Sommerville. *Software Engineering*. International Computer Science Series. Addison Wesley, 2006.
- [22] E. B. Swanson. The dimensions of maintenance. In *Int. Conf. on Software Engineering*, pages 492–497, San Francisco, California, 1976.
- [23] J. Whittle, P. Sawyer, N. Bencomo, B. H. Cheng, and J.-M. Bruel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *Int. Conf. on Req. Engineering*, pages 79–88, Atlanta, Aug. 2009.
- [24] P. Zave and M. Jackson. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6:1–30, 1997.