On the Similarity between Requirements and Architecture

Remco C. de Boer and Hans van Vliet

The Journal of Systems and Software, Vol. 82, Issue 3, March 2009, pp. 544–550

On the Similarity between Requirements and Architecture

Remco C. de Boer *, Hans van Vliet

VU University Amsterdam, Dept. of Computer Science, De Boelelaan 1081a, 1081HV Amsterdam, the Netherlands

Abstract

Many would agree that there is a relationship between requirements engineering and software architecture. However, there have always been different opinions about the exact nature of this relationship. Nevertheless, all arguments have been based on one overarching notion: that of requirements as problem description and software architecture as the structure of a software system that solves that problem, with components and connectors as the main elements.

Recent developments in the software architecture field show a change in how software architecture is perceived. There is a shift from viewing architecture as only structure to a broader view of 'architectural knowledge' that emphasizes the treatment of architectural design decisions as first-class entities. From this emerging perspective we argue that there is no fundamental distinction between architectural decisions and architecturally significant requirements. This new view on the intrinsic relation between architecture and requirements allows us to identify areas in which closer cooperation between the architecture and requirements engineering communities would bring advantages for both.

Key words: Software architecture, requirements, architectural knowledge

1 Introduction

The relation between requirements and software architecture has long been subject to debate. As early as 1994, at the first international conference on requirements engineering, a discussion panel shed its light on the role of software

* Corresponding author. Tel.: +31 20 59 87767; fax: +31 20 59 87728
Email addresses: remco@cs.vu.nl (Remco C. de Boer), hans@cs.vu.nl (Hans van Vliet).

architecture in requirements engineering. In their position papers (Shekaran et al., 1994), all panel members in one way or another hinted at a fundamental distinction between requirements engineering and software architecture. Garlan discussed the difference between problem space and solution space; Jackson contrasted application domain with machine domain; Mead stated that the architect represents the software developer's viewpoint and implies that the requirements engineer represents the customer's viewpoint; Potts distinguished between 'what' and 'how'; Reubenstein defined requirements as an index into solutions, and opposes an idealized architecture to the as-built architecture; Shekaran – like Garlan – distinguished also between problem and solution, but argues in addition that requirements engineering is somewhere in between the two.

Ever since this panel discussion, many have tried to find approaches that bridge the gap between requirements and architecture and integrate the two. Over the years there have been numerous attempts to find such approaches, attempts that have even briefly sparked the dedicated STRAW (software requirements to architectures) workshop series. Those attempts have resulted in many approaches – including the Twin Peaks model (Nuseibeh, 2001), problem frames (Jackson, 2001), and the CBSP approach (Medvidovic et al., 2003) – that have increased our collective awareness of the relation between requirements and architecture. However, this relation is invariably characterized based on the *distinction* between requirements and architecture; a gap that needs to be bridged, often in terms of requirements representing the problem and architecture being the solution.

A problem with this focus on distinction is that the fuzzy line between what is called 'requirements' and what is called 'architecture' can be arbitrarily drawn. Commonly used criteria to tell requirements and architecture apart include 'what' versus 'how', 'problem' versus 'solution', and (a more pragmatic distinction we found to be used in industry) 'determined before' versus 'determined after the contract with the customer has been signed'. Those criteria all carry a notion of 'already fixed' versus 'what remains to be done'. It is nothing new that in software development a clear line between 'fixed' and 'to be done' is impossible to define. Swartout and Balzer's discussion on this 'inevitable intertwining of specification and implementation' (Swartout and Balzer, 1982) holds equally well for the inevitable intertwining of requirements and architecture.

Based on an emerging view on architecture, we hypothesize that there is no fundamental difference between what we call requirements and what we call architecture. Or, to put it more precisely, that architecturally significant requirements are in fact architectural design decisions, and vice versa. We understand that this is a provocative hypothesis that needs further corroboration. In the remainder of this article we show how we arrive at this claim, as well

as some of its implications.

2 Where do Problems End and Solutions Begin?

The ongoing discussion about the relation between requirements and architecture has undoubtedly been influenced by the prevailing view on both fields. In requirements engineering, the prevailing view tends to be one of requirements engineering as problem analysis (cf. Jackson's problem frames, but also goal-oriented approaches such as KAOS). The traditional view on architecture, on the other hand, is one of architecture as solution structure, usually described in boxes-and-arrows type of diagrams. This traditional structure-oriented view is currently shifting to a more knowledge-oriented view.

2.1 Requirements Engineering as Problem Analysis

Requirements engineering is usually seen as focusing on the problem domain, as opposed to the solution domain that architecture belongs to. A prototypical example of requirements engineering as problem analysis, Jackson's problem frames are an increasingly popular requirements engineering instrument for problem analysis and problem decomposition. The problem frames framework emphasizes a thorough understanding of the problem before any solutions are considered.

Jackson discusses the difference between statements on fixed and chosen (or desired) properties in relation to problem analysis by referring to the mood in which statements are expressed (Jackson, 2001). To this end, in his problem frames framework he uses the term 'indicative' for given, objective truths and 'optative' for chosen options. Problem analysis involves specifying the problem domain, the requirements, and the machine specification. The problem domain is the part that is fixed; it is the part of the world where the problem is located which has indicative properties that one is relying on, such as the behaviour of of a car on a road. The chosen parts are the requirements and the machine specification. The requirements are "optative descriptions of what the customer would like to be true in the problem domain", for example [1] the correspondence of a speedometer's display with the current speed of the car; the machine specification is an "optative description [of] the machine's desired behaviour at its interfaces with the problem domain", for instance the way in which a computer receives pulses from a rotating wheel and sets the speedometer's display accordingly.

[1] see (Jackson, 2001) for details on this and more examples.

The limitation of a machine specification to the behaviour at the machine's interfaces suggests a crisp distinction between problem and solution, between requirements and architecture; since the machine domain is considered to provide the solution to the problem at hand, the only options that need to be described are the customer's requirements and the machine's desired behaviour with respect to the problem world. The various solution options, i.e., ways to structure and build the machine, are not yet considered.

However, it has been recognized that problem analysis cannot be completely separated from the considered solutions, since those solutions themselves may affect the problem world. Rapanotti et al. propose an extension of problem frames, called architectural frames, to represent 'those architectural elements that impact the problem description' (Rapanotti et al., 2004). They demonstrate their approach by defining an architectural frame for the pipe-and-filter architectural style. This style mandates an architecture in which components (the 'filters') have input and output connectors that can be linked to form connections (the 'pipes') through which data flows from component to component. Rapanotti et al. argue that using the pipe-and-filter style for a transformation problem (they use the KWIC problem [2] as an example) introduces new sub-problems regarding input, output, transformation, and scheduling. This modification of the problem world leads to new requirements as well: a number of filters needs to be designed, or reused, to address the original transformation problem, input and output data must be converted to a suitable format, and scheduling must be fair (Rapanotti et al., 2004).

While Rapanotti et al. take the use of a pipe-and-filter architectural style for granted, there are more architectural options applicable to the KWIC problem. Chung et al. show how a customer's needs determine the relative criticality of requirements which can be satisfied through different architectural alternatives (Chung et al., 1999). They discuss a scenario where a KWIC component is to be used in a web-based shopping catalog. The e-shopping vendor's user requirements, such as ease of use, good search time performance, and effective space resource utilization, are translated to system requirements regarding for example interactivity, extensibility, and space performance. The high relative criticality of space performance – the number of items in the catalog is expected to increase rapidly – rules out the use of a pipe-and-filter style for the KWIC component. Another option is to use a shared data architecture,

[2] The 'keyword in context' or KWIC problem is a famous problem in software engineering, particularly due to Parnas' discussion of this problem in the context of modular design (Parnas, 1972). A KWIC index system 'circularly shifts' a given line of text, e.g. a publication title, by repeatedly removing the first word and adding it to the end of the line. The resulting KWIC index is a lexicographically ordered list of all shifted lines, in which one can easily find a title when only part of the title is known.

which would meet the required space performance but is ruled out because it fails to meet the extensibility requirements. Instead, Chung et al. conclude that, taking all requirements and criticality values into account, for this particular scenario an implicit invocation style would suit the KWIC component best – even though it doesn't score as good on space performance as a shared data architecture. Hence there is not a one-on-one mapping from a particular problem and/or requirement to a particular architectural solution.

In summary, although the idea of requirements engineering as problem analysis – separated from solution considerations – seems conceptually clean, in reality this separation doesn't hold true. There is a rather intricate interplay between problem and solution. Choices for particular solution directions involve trade-offs that favour certain requirements over others. The choice for a certain solution impacts not only which requirements can be satisfied, but – perhaps even more important – also which ones cannot be satisfied. At the same time, the choice for a particular solution may introduce new (sub)problems and hence new requirements.

2.2 Software Architecture: Beyond Solution Structure

Until recently, the view on architecture has been relatively stable. Architecture was considered to be the (high-level) structure of a software-intensive system, usually in terms of components and connectors. As of the early 2000s, part of the architecture field is moving from this structure-oriented view on architecture to a more knowledge-oriented view, in which design decisions in particular are treated as first-class entities (de Boer and Farenhorst, 2008). As a result, 'the architecture' is no longer solely regarded as the solution structure, but also – occasionally even *instead* – more and more as the set of design decisions that led to that structure. The actual structure, or architectural design, is merely a reflection of those design decisions.

Architectural design decisions are not only directly related to requirements, but they can also be related to other architectural design decisions. The ontology for architectural design decisions constructed by Kruchten (Kruchten, 2004) aims among others to formally describe such relations. Possible relationships between design decisions include constraints, conflicts, and alternatives. For example, the decision to use J2EE constrains the decision to use JBoss (a J2EE application server), and conflicts with the alternative decision to use dotNet. Similarly, the decision to use a pipe-and-filter style constrains the decision to select or design individual filters. Although the impact of the decision on the problem world is not treated explicitly, this is the same kind of mechanism that we saw in Section 2.1; architectural design decisions may introduce new problems to be solved by subsequent decisions.

By breaking down architecture design into individual design decisions, the emerging knowledge-oriented perspective embodies a new way of how we perceive architecture design. It shifts the focus to the process of getting to the architecture design instead of focusing only on the result in terms of components and connectors. It consequently puts more emphasis on the rationale of an architectural design. It also exposes, again, the fuzziness of the 'problem vs. solution' distinction, this time from a solution perspective.

2.3 Problem vs. Solution: A False Dichotomy?

Many requirements can be posed that do not play a role at all at the architectural level, for instance the requirement to use the metric system for a car's speedometer, i.e., display a car's speed in km/h and not mph. We will not consider such requirements, but we limit our discussion to requirements that are 'architecturally significant', in other words requirements that influence the architecture. This is a natural limitation, since we intend to explore the relation between requirements and architecture. Therefore, in the remainder of this article we focus on architecturally significant requirements (ASRs) and architectural design decisions (ADDs) as first-class entities from requirements engineering and architecture design, respectively.

We have already seen that the distinction between problem and solution is at best fuzzy. In Chung's example paraphrased in Section 2.1, for instance, the indicative problem domain properties of a rapid increase of catalog items and limited storage space (implicitly assumed by the original authors) lead to the optative statement that the available space should be effectively used. This statement, clearly an architecturally significant requirement, in turn impacts the eventual choice of architectural style. Note that for the same indicative properties a different requirement could have been phrased, for example the use of scalable storage space. In that case, yet another architectural style might have been more appropriate, or a pipe-and-filter style might have been the best choice after all.

The fuzziness between problem and solution does not merely play a role at a theoretical level, as can be seen from various reports from industry. For example, Poort and De With argue that requirements conflicts they encounter in practice necessarily "arise from limitations in the solution domain" (Poort and de With, 2004). Along the same line, Kozaczynski observes that "in practice key requirements are not fully understood until the architecture is baselined" (Kozaczynski, 2002). And after analyzing five industrial software architecture design methods, Hofmeister et al. conclude that all five methods proceed non-sequentially because "the inputs (goals, constraints, etc.) are usually ill-defined, and only get discovered or better understood as the ar-

chitecture starts to emerge", and that while "most architectural concerns are expressed as requirements on the system, [...] they can also include mandated design decisions" (Hofmeister et al., 2007). Finally, Savolainen and Kuusela present a system design framework in which they suggest to mix design and requirements specifications when appropriate since "all design details that are not optional can be treated as highly constrained requirements" (Savolainen and Kuusela, 2002).

Apparently the choice of required behaviour given a particular problem can have tremendous impact on the architecture of the machine that deals with that problem. Moreover, the decision to use a particular architectural style again impacts the problem world, where it may introduce new requirements, as well as subsequent ADDs. Hence both ASRs and ADDs appear to live in a grey area between the desired behaviour in the problem world and the desired properties of the machine to be constructed. Their origin and their effect are not confined to either problem or solution.

3 Why Architectural Requirements and Design Decisions Are Really The Same

The difficulty to distinguish between ASRs and ADDs is perhaps not all that strange if we consider that both are optative statements, or decisions, that embody what one wants to achieve. Those decisions may constrain, or may themselves be constrained by, other decisions. This means that every optative decision based on an indicative problem setting has itself indicative properties from the perspective of subsequent decisions, as depicted in Fig. 1. In other words, the decision is reflected in, and may even become part of, the problem world. Some of those decisions we call 'requirements', other 'architectural design decisions', but there is no fundamental difference between the two.

To illustrate this point, let us consider a system that consists of a dedicated machine, say a package router (cf. (Jackson, 2001)), that we need to build. If the hardware configuration has already been determined, it is part of the indicative problem world and its physical layout of pipes and switches would co-determine and constrain the requirements and subsequent architectural design of the software. If, on the other hand, the hardware configuration has yet to be decided upon, architectural design decisions would have to be made to determine the architecture of the software as well as the architecture (or physical layout) of the hardware. Obviously, decisions on the layout of the physical pipes and switches would still constrain the feasibility of requirements and the architecture for the system's software. The only difference with a predetermined hardware configuration is that in this new situation the architecture of the hardware is also chosen (optative), rather than given (indicative). This

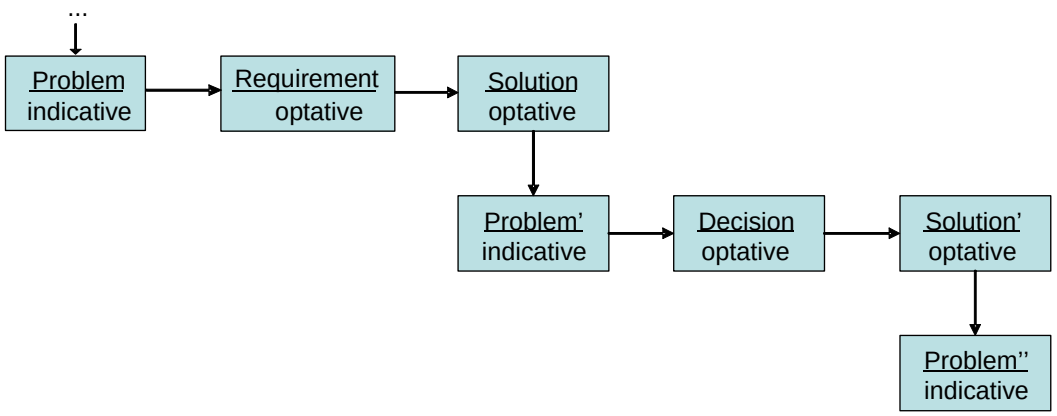

Figure 1. Indicative properties of optative decisions.

means that in the former situation the hardware is part of the problem, and its use a requirement, whereas in the latter situation the hardware is part of the solution and has effects in the problem domain. Similar shifts in indicative and optative properties occur for example in the selection of COTS components or the use of a reference architecture or framework, where new requirements and architectural design decisions both need to cope with the predefined architectural properties of the selected software.

Both ASRs and ADDs have similar effects on the development of software: they indicate preferences for the direction in which the software should develop and prune the design space by ruling out undesired alternatives. This similarity may only be apparent if one does not regard architecture merely as structure, but embraces the view that architectural design decisions are an important element of architectural knowledge.

Relations between architectural design decisions can be modeled as a 'decision loop', in which ADDs introduce new design issues for which new ADDs need to be taken. The theoretical underpinning of such a decision loop can be found in (de Boer et al., 2007). With this decision loop depicted in Fig. 2 in mind, we can explain the similarity of ASRs and ADDs from an architectural knowledge point of view.

Suppose that we have a situation in which we need to build a KWIC component that is highly extensible and has effective space performance, basically the same situation that we discussed in Section 2.1. Those requirements ask for several decisions to be taken. In other words, the requirements introduce one or more *decision topics*, most notably 'how to limit the amount of data stored'. This decision topic can be addressed through various alternatives, such as 'shared data', 'abstract data types', and 'implicit invocation'. When one of these alternatives, for instance 'implicit invocation', is chosen (note that the way in which the alternatives are ranked depends on various concerns, including 'extensibility') that alternative becomes an *architectural design decision*

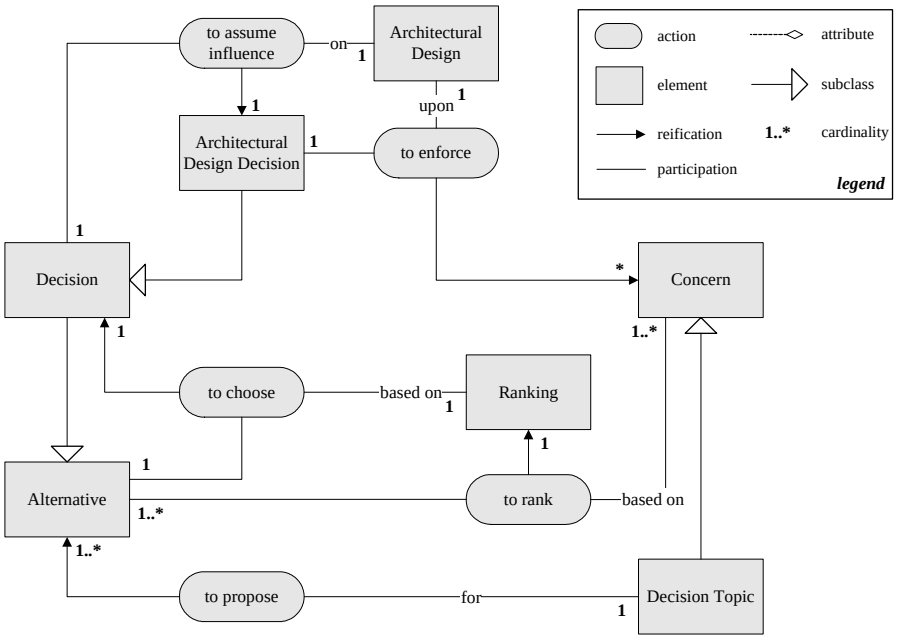

Figure 2. Architectural design decision loop

which immediately leads to new decision topics (e.g., 'how to distribute events to components').

Furthermore, the requirement that the KWIC component has effective space performance may stem from a higher-level (possibly implicit) *concern*, in this case the concern that the component can handle an increasing amount of data. Indeed, the requirement that the KWIC component must have effective space performance could itself be regarded as an architectural decision that addresses the requirement 'must cope with an increasing amount of data'; it tells *how* the system should cope with that increase, although another alternative (e.g., 'scalable data storage') could have been selected as well.

In short, an architectural design decision (or 'solution') for a requirement (or 'problem') inevitably introduces new 'problems'. If the problem is 'the system must be able to cope with an increasing amount of data' and the solution is 'by means of effective space performance' then one of the new problems is 'the amount of data stored must be limited'. In this decision loop, concerns (part of the problem) lead to decisions (part of the solution) which lead to new concerns (again, part of the problem). In the example above, effective space performance could just as well be regarded a requirement as an architectural design decision, depending on who you ask – or rather: depending on who has coined it, the requirements engineer or the architect. In any case, the need to cope with an increasing amount of data is the reason (or rationale) behind the choice for effective space performance. Note that the decision loop described here may extend beyond the architecture design phase; some architectural decisions may be taken earlier, some later.

As in the architectural design decision loop, the idea of 'rationale' plays a role in requirements engineering as well, for instance in 'goal-oriented requirements engineering' or GORE (van Lamsweerde, 2001). Given any goal, by asking *why* that goal is needed higher level goals can be found. The other way around, by asking *how* a system may help satisfy a goal, that goal may be refined into new – lower level – goals. Eventually, goals are refined to the level where every goal is realizable, either in the environment or in the software. Those latter goals are the requirements, hence by means of goal refinement a hierarchical structure of goals and requirements develops. In GORE, the decision loop described above appears in a requirements engineering context. The same 'how' and 'why' questions used in goal refinement can be asked for concerns and decisions in the decision loop. Indeed, even though architectural design decisions are not an explicit part of the GORE refinement process, Van Lamsweerde maintains that "high-level architectural choices are already made during that process" since "for each [refinement] decisions have to be made which in the end will produce different architectures" (van Lamsweerde, 2003).

We have now discussed various examples of problem and solution influencing each other, and how this leads to ambiguity regarding whether to call something a requirement or a design decision. But haven't we unnecessarily and unrealistically complicated things by questioning the distinction between problem and solution in the first place?

As it turns out, there seems to be evidence from the field of psychology that shifting between problem and solution really occurs in human reasoning. Holyoak and Simon discuss this bidirectional decision making, in which 'the distinction between premises and conclusions is blurred' (Holyoak and Simon, 1999), much like our observation of the blurred distinction between ASR and ADD. This type of decision making is thought to play a role in the presence of complexity, conflict, and ambiguity; characteristics that accompany most if not all software engineering situations in which architecture design and requirements engineering would play a pivotal role.

4 Architectural Statements and the Magic Well

In a way, architecturally significant requirements and architectural design decisions seem to accumulate in some kind of a 'magic well'. Observers peering into the well see what they wish for. People wishing to find architecturally significant requirements will see architecturally significant requirements; people looking for architectural design decisions will find architectural design decisions. Hence, this magic well warps the way architectural statements (i.e. ASRs and ADDs) are perceived depending on the observer's perspective, as shown in Fig. 3.

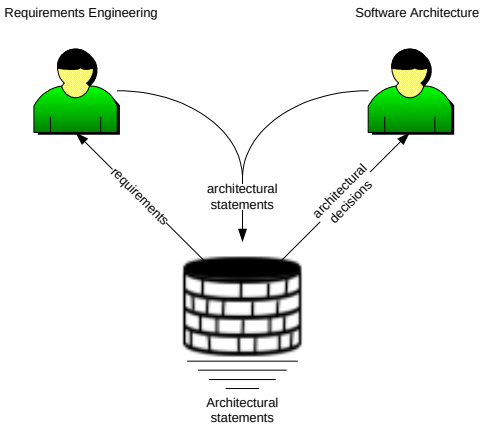

Figure 3. Magic Well: Different perceptions from different angles.

Table 1
Methods and techniques: Requirements engineering vs. architecture.

Requirements Engineering	**Magic Well**	**Architecture**
Elicitation Negotiation	Creation of statements	Decision making Trade-off analyses
Specification	Drop statements in well	Design
Validation	Compare well contents with reality	Assessment
Documentation	Write down well contents	Description
Requirements management	Structure well contents	Architectural knowledge management

The well does not only determine how we perceive statements, but also affects methods and techniques. Table 1 shows requirements engineering and software architecture counterparts of actions that involve the well.

Both fields have their own particular methods and techniques to work with the magic well. For instance, requirements engineering and architecture both have their own standardized approaches to write down what is in the well. IEEE Std. 830 and IEEE Std. 1471 are well known examples of recommended practices for requirements documentation and architecture description respectively.

Dropping freshly created statements in the well is akin to what requirements engineers call specification, and what architects call design. Any elicited requirement or architectural design decision has to be made explicit in one way or another. Not doing so will undoubtedly result in the architectural statement being ignored, or even forgotten, somewhere down the line. Both fields have their own preferred (not necessarily disjunct) sets of techniques for ex-

pression, such as natural language text, ER diagrams, UML diagrams, and other box-and-line diagrams.

Elicitation in requirements engineering puts a great deal of emphasis on elicitation techniques, such as interviews, focus groups, use cases, and prototyping. Through negotiation, the priority or relative weight of each requirement is determined. Architecture, on the other hand, focuses more on choosing the right alternative and making trade-off analyses. Requirements are not necessarily left implicit, but are processed less formally in architecture. The Architecture Business Cycle (Bass et al., 2003), for instance, clearly states the importance of addressing stakeholders and their requirements and concerns throughout the architecting process, but does not provide any methodological pointers as to how to elicit those requirements. Note that in both requirements elicitation and architectural decision making a certain amount of creativity is involved; like requirements, architectural solutions may need to be discovered or invented first.

In both requirements engineering and architecture, validation is an integral part of the process. Architecture validation is usually called assessment or evaluation. The architecture community has devised various approaches to architecture assessment that either focus on a single architectural quality such as modifiability (e.g., ALMA (Bengtsson et al., 2004)) or may consider various qualities (e.g., SAAM (Kazman et al., 1994)) and the tradeoffs between them (e.g., ATAM (Kazman et al., 1999)). Requirements validation typically follows a somewhat less formal approach with techniques like inspections and reviews. Although the details of the methods differ, they also have much in common. Both have well-defined process descriptions of how to go about in a validation. But in practice, organizations often use a cafetaria-like approach, in which they borrow method snippets that best fit the situation at hand. Methods in both architecture assessment and requirements validation are often scenario-based. And the results of both surpass the mere identification of errors.

Roeller et al. compared architectural design decisions with material floating in a pond. "When not touched for a while, they sink and disappear from sight" (Roeller et al., 2006). If one just drops statements in the well and leaves them there, this is exactly what will happen. Management of the architectural statements in the well has therefore received considerable interest from the requirements engineering community and, more recently, the architecture community. Both requirements management and architectural knowledge management have to do with regarding the well not as an accumulation of statements, but as an information repository. Both aim to impose a structure on the well's contents that captures the connections between individual requirements or architectural design decisions.

4.1 Cross-fertilization

From Table 1 we can try to identify areas in which the architecture community could benefit from a closer inspection of the state of the art in requirements engineering and vice versa. Apart from high-level models such as the Architecture Business Cycle (ABC), for instance, the architecture community seems to have hardly any standardization or best practices on elicitation. This may not be a big surprise, given that the magic well probably leads us to believe that elicitation (of requirements) is not the architect's job. But on the other hand, the ABC puts quite a bit of emphasis on interaction with stakeholders and understanding their requirements. Stakeholder concerns and business goals play a major role for software architects as well, and using approaches proven and tested in requirements engineering might pay off.

The most interesting area for cross-fertilization, however, is probably the area of management. The emphasis on architectural design decisions and the subsequent focus on architectural knowledge and architectural knowledge management are a relatively recent development in the architecture community. While this has led to many promising initiatives over the short time span of only a few years, the requirements engineering community has also been working on similar issues that the architecture community now encounters. Both fields encounter challenges in areas such as traceability, consistency, evolution, and rationale management.

Current initiatives in architectural knowledge management mainly focus on the development of frameworks and models to capture architectural knowledge (Ali Babar et al., 2007). Many architectural knowledge management initiatives seem to have some overlap with approaches in requirements management.

In requirements engineering, goals play a role in many areas related to architectural knowledge management, including traceability, conflict detection and resolution, and exploration of design choices (Rolland and Salinesi, 2005). The requirements engineering community has devised several methods to model goals, such as i* (Yu, 1997) and KAOS (van Lamsweerde, 2001).

An example of a still relatively unexplored area in requirements engineering is that of requirements interdependencies. Dahlsted and Persson sketch a research agenda with four major research areas (Dahlstedt and Persson, 2005): 1. What is the nature of requirements interdependencies?; 2. How can we identify requirements interdependencies?; 3. How can we describe requirements interdependencies?; and 4. How do we address requirements interdependencies in the software development process? In architectural knowledge management, interdependencies (between architectural design decisions) play an important

role as well. The ontology of ADDs (Kruchten, 2004) that Kruchten proposes, for instance, tries to address some of the questions above from an architecture perspective. Especially in those management areas where both communities are just beginning to find answers, close collaboration and regular exchange of research results between the two communities should be advantageous to all.

5 Towards Tighter Collaboration

We have posed in this article a statement that many may find provocative, or even controversial. We claim that architectural design decisions and architecturally significant requirements are really the same; they're only being observed from different directions. Even though it may be controversial, our claim is intended to be constructive since we feel it may open the road towards tighter collaboration between the requirements and architecture fields.

Currently, the challenges in requirements management and architectural knowledge management are approached as if there are two separate 'information wells'. Both communities perform research on comparable challenges without paying too much attention to what the others are doing. If we recognize and acknowledge that both communities are in fact looking at the same 'magic well' from different angles, we open the door for tighter collaboration between the fields as well as reuse of each other's research results.

Although there are still many open issues in requirements management, when compared to architectural knowledge management that field is much more mature. This is not to say that the current work in researching architectural knowledge management is in vain. On the contrary, innovations in architectural knowledge management might have a beneficial impact on requirements management problems as well. However, we as a software architecture community can probably learn a great deal from our colleagues in requirements engineering.

The magic well has yet another implication: software architecture is not merely the domain of the architect. Each architecturally significant requirement is already a decision that shapes the architecture. Nevertheless, because of their complementary approaches we still need both requirements engineers and architects; we cannot do without either of them. Requirements engineering and software architecture have considerable overlap, but also use different perspectives. We should, therefore, further align our tools, methods, and techniques by focusing more on our common ground, rather than on our differences. Further exploring and exploiting the *commonalities* between architecturally significant requirements and architectural design decisions would serve that goal.

Acknowledgment

This research has been partially sponsored by the Dutch Joint Academic and Commercial Quality Research & Development (Jacquard) program on Software Engineering Research via contract 638.001.406 GRIFFIN: a GRId For inFormatIoN about architectural knowledge. The authors thank Rik Farenhorst for sharing some well thought out ideas. We thank the anonymous reviewers for their suggestions and comments.

References

Ali Babar, M., de Boer, R. C., Dingsøyr, T., Farenhorst, R., 2007. Architectural Knowledge Management Strategies: Approaches in Research and Industry. In: Second Workshop on SHAring and Reusing architectural Knowledge - Architecture, rationale, and Design Intent (SHARK-ADI). Minneapolis, MN, USA.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice, 2nd Edition. SEI Series in Software Engineering. Addison-Wesley Pearson Education, Boston.

Bengtsson, P., Lassing, N., Bosch, J., van Vliet, H., 2004. Architecture-level modifiability analysis (ALMA). Journal of Systems and Software 69 (1-2), 129–147.

Chung, L., Gross, D., Yu, E., 1999. Architectural Design to Meet Stakeholder Requirements. In: Donohoe, P. (Ed.), First Working IFIP Conference on Software Architecture (WICSA). pp. 545 – 564.

Dahlstedt, Å. G., Persson, A., 2005. Requirements Interdependencies: State of the Art and Future Challenges. In: Aurum, A., Wohlin, C. (Eds.), Engineering and Managing Software Requirements. Springer-Verlag, Berlin Heidelberg, pp. 95–116.

de Boer, R. C., Farenhorst, R., 2008. In Search of 'Architectural Knowledge'. In: 3rd Workshop on SHAring and Reusing architectural Knowledge (SHARK). Leipzig, Germany.

de Boer, R. C., Farenhorst, R., Lago, P., van Vliet, H., Clerc, V., Jansen, A., 2007. Architectural Knowledge: Getting to the Core. In: Third International Conference on Quality of Software-Architectures (QoSA). Vol. 4880 of LNCS. Springer, pp. 197–214.

Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A., America, P., 2007. A general model of software architecture design derived from five industrial approaches. Journal of Systems and Software 80 (1), 106–126.

Holyoak, K. J., Simon, D., 1999. Bidirectional Reasoning in Decision Making by Constraint Satisfaction. Journal of Experimental Psychology: General 128 (1), 3–31.

Jackson, M., 2001. Problem Frames: Analyzing and structuring software development problems. ACM Press Books, Addison-Wesley.

Kazman, R., Barbacci, M., Klein, M., Carrièrre, S. J., Woods, S. G., 1999. Experience with Performing Architecture Tradeoff Analysis. In: 21st International Conference on Software Engineering (ICSE). Los Angeles, California, United States, pp. 54–63.

Kazman, R., Bass, L., Webb, M., Abowd, G., 1994. SAAM: A Method for Analyzing the Properties of Software Architectures. In: 16th International Conference on Software Engineering (ICSE). Sorrento, Italy, pp. 81–90.

Kozaczynski, W., 2002. Requirements, Architectures and Risks. In: 10th Anniversary Joint IEEE International Requirements Engineering Conference (RE). p. 6.

Kruchten, P., 2004. An Ontology of Architectural Design Decisions in Software-Intensive Systems. In: 2nd Groningen Workshop on Software Variability Management. Groningen, NL.

Medvidovic, N., Grünbacher, P., Egyed, A., Boehm, B. W., 2003. Bridging models across the software lifecycle. Journal of Systems and Software 68 (3), 199–215.

Nuseibeh, B., 2001. Weaving Together Requirements and Architectures. IEEE Computer 34 (3), 115–117.

Parnas, D. L., 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM 15 (12), 1053 – 1058.

Poort, E. R., de With, P. H., 2004. Resolving Requirement Conflicts through Non-Functional Decomposition. In: Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA). IEEE Computer Society, p. 145.

Rapanotti, L., Hall, J. G., Jackson, M., Nuseibeh, B., 2004. Architecture-driven Problem Decomposition. In: 12th IEEE International Requirements Engineering Conference (RE). pp. 80–89.

Roeller, R., Lago, P., van Vliet, H., 2006. Recovering Architectural Assumptions. Journal of Systems and Software 79 (4), 552–573.

Rolland, C., Salinesi, C., 2005. Modeling Goals and Reasoning with Them. In: Aurum, A., Wohlin, C. (Eds.), Engineering and Managing Software Requirements. Springer-Verlag, Berlin Heidelberg, pp. 189–217.

Savolainen, J., Kuusela, J., 2002. Framework for Goal Driven System Design. In: 26th Annual International Computer Software and Applications Conference. p. 749.

Shekaran, C., Garlan, D., Jackson, M., Mead, N. R., Potts, C., Reubenstein, H. B., 1994. The Role of Software Architecture in Requirements Engineering. In: First International Conference on Requirements Engineering (ICRE). pp. 239–245.

Swartout, W., Balzer, R., 1982. On the Inevitable Intertwining of Specification and Implementation. Communications of the ACM 25 (7), 438–440.

van Lamsweerde, A., 2001. Goal-Oriented Requirements Engineering: A Guided Tour . In: 5th IEEE International Symposium on Requirements Engineering (RE). pp. 249–263.

van Lamsweerde, A., 2003. From System Goals to Software Architecture. In: Bernardo, M., Inverardi, P. (Eds.), Formal Methods for Software Architectures. Vol. 2804 of LNCS. Springer-Verlag, pp. 25–43.

Yu, E. S. K., 1997. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In: Third IEEE International Symposium on Requirements Engineering (RE). pp. 226–235.