Characterizing Architecturally Significant Requirements

Lianping Chen, Muhammad Ali Babar, and Bashar Nuseibeh
Lero—the Irish Software Engineering Research Centre, University of Limerick, Ireland
IT University of Copenhagen, Denmark

1. Introduction

At the heart of any engineering discipline is the interplay between problem and solution development. In software engineering, the effectiveness of a software solution is determined with respect to a problem, yet the nature of the problem and its scope may be very dependent on what solutions already exist or what solutions are plausible and cost-effective. This iterative and concurrent development and trade off between software problems and solution characterise the 'twin peaks' model of software development [1].

For the development of many large scale software intensive systems, the software architecture is a fundamental part of a software solution [2]. It has a significant impact on software quality and cost. However, not all of a system's requirements have equal impact on the software architecture [3]. In many cases, only some requirements actually determine and shape a software architecture [4], and we term these *Architecturally Significant Requirements* (ASRs). If ASRs are wrong, incomplete, inaccurate, or lack details, then a software architecture based on these is also likely to contain errors.

Unfortunately, in practice, ASRs are frequently not expressed nor communicated effectively to architects, to enable them to make informed design decisions [4]. This is partly due to a lack of an authoritative, evidence-based discourse characterising ASRs.

To address this, we carried out an empirical study using Grounded Theory [5] to help characterize ASRs. The study involved interviews with 90 practitioners who together have more than 1448 years of accumulated software development experience in more than 500 organizations.

Based on the study's results, this article presents an evidence-based framework for systematically characterizing ASRs. Our findings may help practitioners better understand and deal with ASRs. Our findings also provide researchers with a framework for discussing, and conducting further research on ASRs, and can inform researchers' development of technologies for dealing with ASRs. Finally, our findings also enrich our understanding of requirements and architecture interactions, allowing the 'twin peaks' to move from aspiration to reality.

2. Methodology and Research Design

Our research method was Grounded Theory (GT) [5], which provides a systematic process of generating theory from data. GT has gained popularity among software engineering researchers who have reported its effectiveness for topics that lack empirical research [6]. GT recommends avoiding the formulation of research questions upfront, but rather a general area of interest is selected for research. We chose to explore ASRs as an area of research.

The data we used in our study was collected by conducting telephone and email interviews with 90 practitioners, who were recruited through researchers' personal connections and social networks. Figure 1 summarizes the demographics of the participants who came from 4 countries (USA: 58%, India: 28%, the Netherlands: 13%, and UK: 1%).

Figure 1b shows that the participants represented a diverse set of industries. The participants had worked, on average, for 7 organizations during their career. The total number of distinct organizations the participants had worked for was more than 500. The size of the organizations they worked for ranged from very small, through to very large, as shown by Figure 1c.

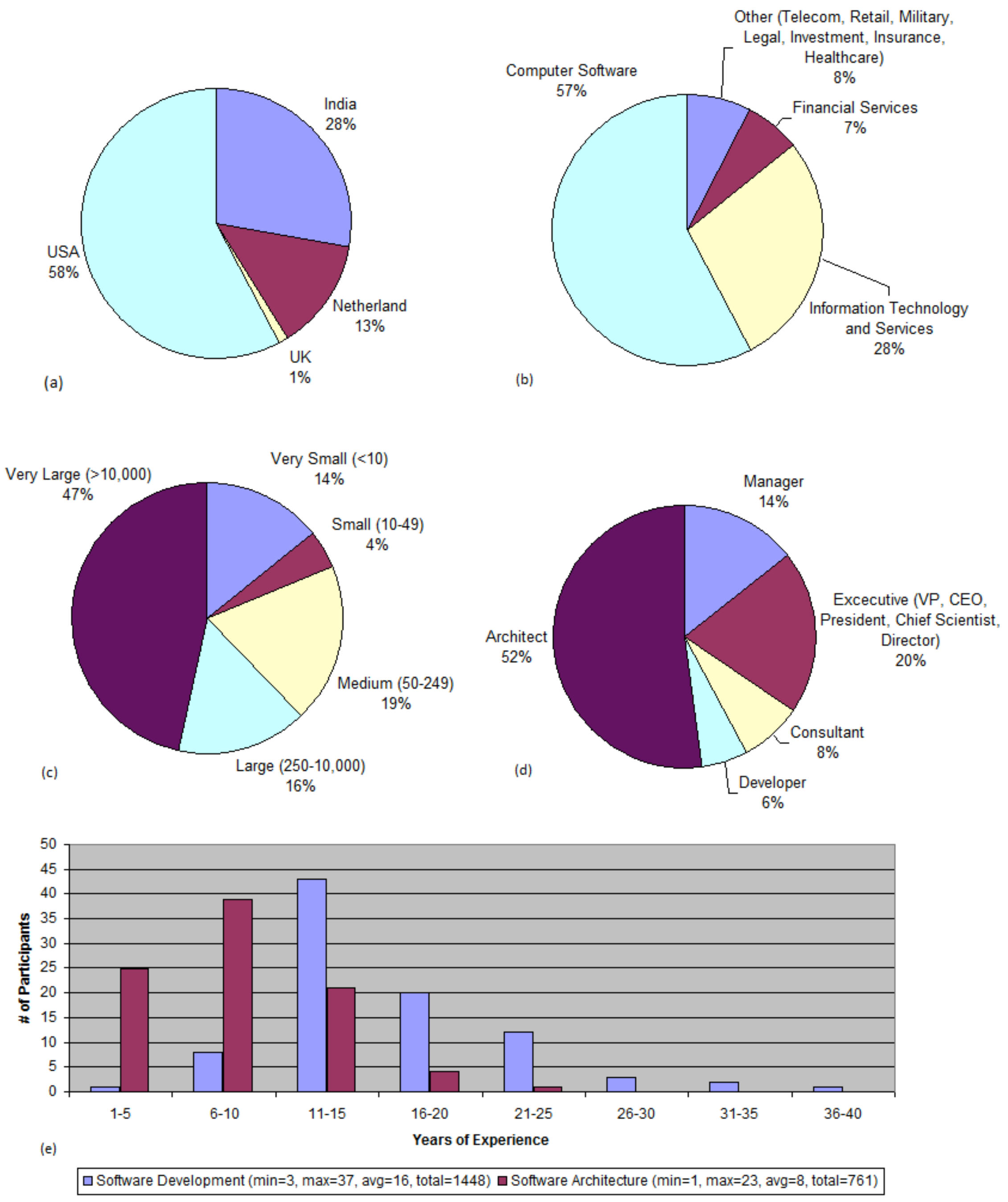

Figure 1. Demographics of participants: (a) participants' distribution over countries, (b) participants' distribution over industries, (c) participants' distribution over organization sizes, (d) participants' job positions, and (e) participants' experience in software development and architecture.

Given the focus of our research on understanding the impact of requirements on architectures, we decided to involve only those professionals who had experience of working with software architectures in their current or previous roles. Whilst the majority of the participants we interviewed were software architects, some were executives, managers, consultants, and developers. None of our participants identified themselves as "requirements engineers", but we observed that consultants often played such roles. As Figure 1d shows, 52% of our participants were architects, followed by 20% executives, 14% managers, 8% consultants, and 6% developers. Together they had 1448 years of accumulated work experience in software development and 761 years in software architecture, as shown by Figure 1e.

Our interviews were conducted in informal conversation style. Most of the conversations kicked off with the following question:

> *From your observations and experiences, what are the characteristics that distinguish ASRs from other requirements? In other words, what, if anything, is it that makes them unique or peculiar?*

Follow up questions sought to clarify or gather more details about the characteristics mentioned by the participants.

For analysing the data, we used qualitative data analysis techniques suggested by GT, such as open coding, constant comparison method, selective coding, and memoing [7]. The data analysis commenced as soon as some data has been collected and continued iteratively and in parallel. We used a tool called NVivo to facilitate the analysis.

Only concepts that were mentioned by at least two participants were included in the final findings. Once concepts earned their way into the theory, we did not discriminate between them based on their frequencies; rather we focused on the logical relationships among concepts, as recommended by GT.

3. Characteristics of Architecturally Significant Requirements

A framework of characteristics of ASRs emerged from our analysis. Figure 2, shows the framework, which consists of four sets of characteristics: Definition, Descriptive, Indicators, and Heuristics. We present each of these components in the following subsections.

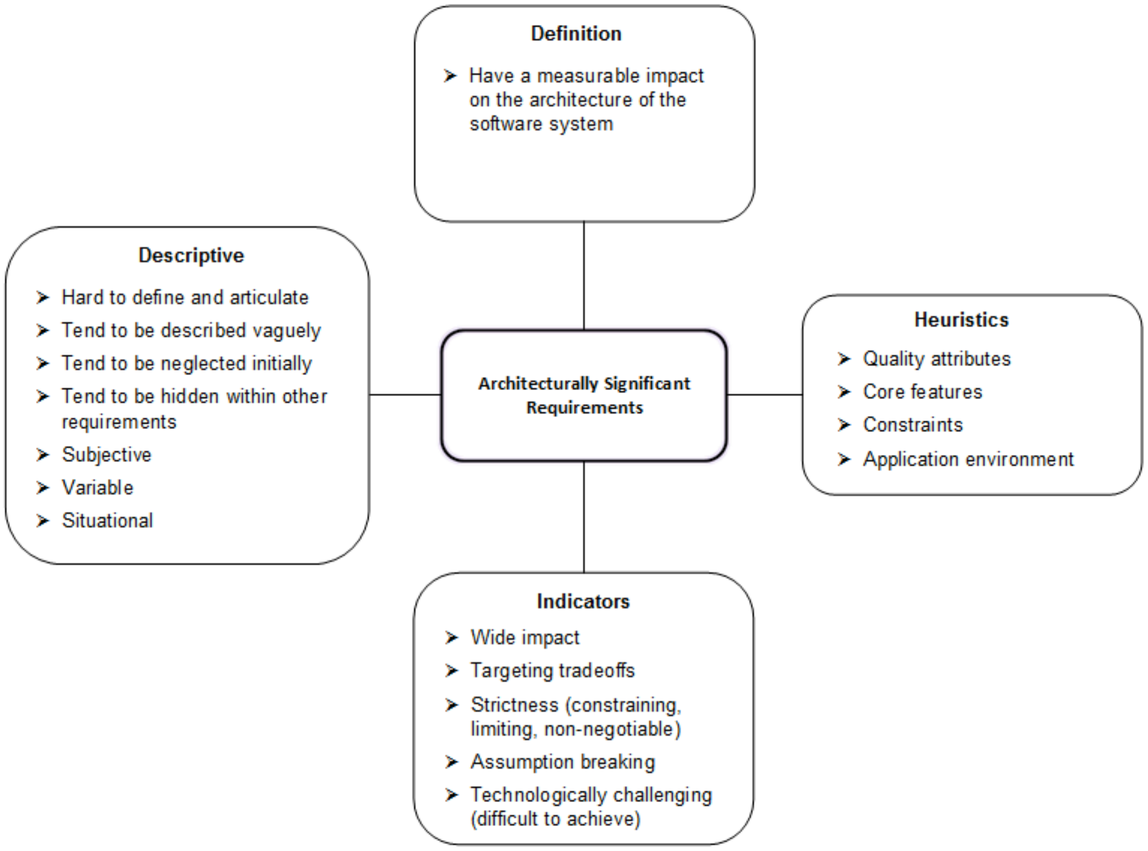
Definition
➤ Have a measurable impact on the architecture of the software system
Descriptive
➤ Hard to define and articulate
➤ Tend to be described vaguely
➤ Tend to be neglected initially
➤ Tend to be hidden within other requirements
➤ Subjective
➤ Variable
➤ Situational
Architecturally Significant Requirements
Heuristics
➤ Quality attributes
➤ Core features
➤ Constraints
➤ Application environment
Indicators
➤ Wide impact
➤ Targeting tradeoffs
➤ Strictness (constraining, limiting, non-negotiable)
➤ Assumption breaking
➤ Technologically challenging (difficult to achieve)

Figure 2: A Framework of Characteristics of Architecturally Significant Requirements

3.1 Definition

Architecturally significant requirements are defined as those requirements that have a measurable impact on the architecture of the software system. Essentially, the definition delimits a portion of requirements, the portion that affects architecture of the system in measurably identifiable ways. Do such requirements really exist? Our empirical data gathered appears to confirm this. For example, requirements stating that "temperature should be displayed in Celsius not Fahrenheit on this web page" were not regarded as architecturally significant, whereas a requirements stating that "the system should provide five nines (99.999%) availability" was usually regarded as architecturally significant.

The criterion "significant impact on the architecture" that demarcates these requirements was the overall distinguishing characteristic of ASRs.

"Significant" is a key term in our definition. What does it mean? Ultimately it is measured by high cost of change. This cost can be monetary or non-monetary (e.g., time, resources, reputation, and opportunity cost). What cost is considered as high? The answer is invariably project-specific. The same cost can be high for a small budget project, but low for a large one. Other cost measures can be entirely qualitative, but still identifiable in their ability to distinguish ASRs.

3.2 Descriptive Characteristics

ASRs are often hard to define and articulate, tend to be expressed vaguely, tend to be neglected initially, tend to be hidden within other requirements, are subjective, variable, and situational. One may argue that other requirements may also demonstrate these characteristics. However, the architectural significance of these requirements made the manifestation of these characteristics unique and challenging for ASRs. We describe each of them below.

3.2.1 Hard to Define and Articulate

ASRs are usually hard to define and articulate by stakeholders. As one participant summarized, "users usually find it difficult to articulate these requirements *[ASRs]*, as many of them are about abstract and general concepts."

In addition, ASRs are expected to be ready in the early stage of the software development process. At this early stage, customers may not be clear about their exact needs, and some requirements decisions (especially about details) may have not yet been made. This adds to the difficulties of defining and articulating ASRs.

3.2.2 Tend to Be Described Vaguely

ASRs tend to be expressed vaguely by customers and clients. Many architects reported that requirements are too vague to be used for making informed architectural decisions. Vaguely described ASRs usually lead to wrong decisions because architects may make wrong assumptions about the missing details.

For example, users requested the "ability to receive notification about cash flows". Architects made assumption that emails would be acceptable. During detailed design, users explained that they want "real time notification and ability to subscribe to different account topics and it is shown on user interface". This requires Publish-Subscribe, a different architectural style.

3.2.3 Tend to Be Neglected Initially

ASRs tend to be neglected initially. "Typically these requirements *[ASRs]* are overlooked in the early phase of a project", as reported by a participant. ASRs are neglected initially because people are not initially aware of their significant impact on architecture, thus they do not give them sufficient attention. One participant added that "users do not typically have a good understanding of them, people who conduct requirement analysis often do not document them properly... ... users will not ask for them unless you are dealing with a highly tech-savvy group of users." This very often happens for less experienced teams, said by another participant.

Many times, only when the requirements have incurred a high cost, are they recognized as architecturally significant. However, at this late stage, rectify the mistakes can be very costly.

3.2.4 Tend to be Hidden within Other Requirements

ASRs tend to be hidden within other requirements that are not architecturally significant. Following the way people spontaneously express requirements, ASRs are usually not highlighted and emphasized in the (oral or written) descriptions of requirements; they are embedded in the descriptions of other requirements. For example, short phrases like "highly available system of 99.999% uptime" or "fault tolerant" are often briefly mentioned while describing other requirements. These short phrases can have significant impact on architecture.

When ASRs are passed to architects in a way that is hidden within other requirements, the ASRs are usually not sufficiently elaborated and often miss key details that are required for making informed architectural decisions.

3.2.5 Subjective

ASRs tend to be requested/asked subjectively based on opinion instead of facts-based objective decisions. One participant reported that "They *[ASRs requested by customers]* usually contain subjectivity, for example, 'the system should be available for 24×7', whether it *[needs to]* be or not."

This subjectivity can lead to inaccuracy of the requirements conveyed to architects. However, small differences in ASRs can lead to big differences in the resulting architecture.

3.2.6 Variable

ASRs can be variable both over time and space. ASRs can change over time. This usually cannot be avoided completely due to changes that take place in business and technology. As a participant reported, "requirements always change over time, during the project and afterwards." Another participant reported that "technology as defined as hardware, software, and the relatively new inclusion of User Experience, are now changing so quickly that companies are viewing products as a very limited (and short) lifetime before needing to be redesigned to take advantages of those changes."

ASRs can also vary over space. This refers to the variation of ASRs at the same point of time for different but similar software systems that are engineered using a software product line paradigm [8].

3.2.7 Situational

Whether a piece of requirement is architecturally significant depends on situations. As one participant said "a requirement is architecturally significant in one case, while being 'just a requirement' in the other case."

Requirements can be situational with respect to an existing architecture, project context/scope, etc. For example, a requirement can be architecturally significant in a situation where a ‘bad’ architecture is in place, but is not so when the architecture is ‘good’. Another participant reported that: "A simple requirement in a smaller project may not be architecturally significant. Take the same requirement and enhance scope, add multiple interactions, then it could become significant."

A requirement's architectural significance may depend on many factors. So, a definitive judgement on a requirement's architectural significance usually can only be made until the requirement really incurs a high cost or is required by the architects to make architectural decisions. This suggests that we can only say that certain requirements are "likely" to be architecturally significant during requirements gathering. This also implies that finding a definitive list of ASRs is not feasible. ASRs need to be dealt individually.

3.3 *Indicative Characteristics (Indicators)*

Although the cost of change is a measure of significance, getting an accurate cost of a requirement is itself challenging, although cost estimation has been studied extensively [9]. In addition, cost estimation is usually undertaken after requirements have been gathered. However, an estimate of whether a requirement is architecturally significant can be important for guiding the gathering of ASRs.

Do ASRs have some characteristics for us to distinguish them from other requirements without undertaking a full cost estimation? Two sets of characteristics, indicative and heuristic, were identified by our study. In this section, we will present indicative characteristics, which include wide impact, targeting tradeoffs, strictness, assumption breaking, and technically challenge.

3.3.1 Wide Impact

When a requirement has a wide impact on a system, it is usually architecturally significant. This can be in terms of the components, other requirements, the code modules, the stakeholders, and others that are affected by the requirement. Participants noted that "Yes, there are requirements which can be thought of as architecturally significant because those requirements impact the system as a whole immediately". "It has widespread impact across multiple components of the system". "The more broadly a requirement and its resolution can be applied, the more significant it is".

3.3.2 Targeting Tradeoffs

A requirement that targets tradeoff points is usually architecturally significant. A tradeoff point is a point where there is no solution that satisfies all involved requirements equally well, and architects have to select a design option that compromises some requirements and to meet others. The requirements, which offer these tradeoff points, have direct impact on the outcome of the architectural decisions. Thus, they are architecturally significant. One participant reported this as follows:

"The tradeoffs are the weak points - the raw nerves - of the architecture. If a new requirement happens to (unintentionally) target these tradeoffs - these weak points - then probability of it becoming architecturally significant is higher than if it did not target these tradeoffs."

When the requirement targets a tradeoff point, the details, accuracy, and precision of the requirement description become important. For example, a participant reported that half a second difference in the performance requirements of his system can lead totally different design. Thus, in his case, the performance requirement needed to be captured accurately and precisely, otherwise a wrong design choice could be made.

3.3.3 Strictness (Constraining, Limiting, Non-negotiable)

Strictness refers to the situation where a requirement strictly requires a particular design option and cannot be negotiated. When making architectural decisions, the requirements that can be satisfied by multiple design options (or can be negotiated to a form that can be satisfied by multiple design options) will provide the flexibility for the design. The architectural decision will be determined by the requirement that is strict and cannot be satisfied by other design options. One participant reported that "our significant requirements were those that would be the limiting, or defining, characteristics of the product."

3.3.4 Assumption Breaking

When the architecture of the system is being designed, some fundamental assumptions are made, explicitly or implicitly, due to various reasons. When a requirement breaks any of these assumptions, it is architecturally significant. This is because it requires architectural change to accommodate the requirement, as one participant reported: "down the line someday, we meet our assumptions face to face, and a requirement which actually crosses the boundary of that assumption would be the one which changes the architecture."

Judging whether a requirement breaks any of such fundamental assumptions requires good knowledge about the existing architecture of the system. A well organized record of the assumptions and related architectural decisions can make the job easier. This suggests the need for effective tools to mange assumptions and design decisions.

3.3.5 Technologically Challenging (Difficult to Achieve)

If a requirement is difficult to meet or is technologically challenging, this requirement is likely to be architecturally significant. One participant reported that "The uniqueness of these requirements *[ASRs]* to me is ... ... difficulty of achieving. For example latency is very hard to achieve if not thought *[about]* early on."

3.4 Heuristic Characteristics (Heuristics)

Are the indicators described above easy to use? Judging whether a requirement has wide impact, whether a requirement targets tradeoffs, whether a requirement strictly requires a particular design option, whether a requirement breaks existing architectural assumptions, or whether a requirement is difficult to achieve may require substantial knowledge of the solution space.

Requirements engineers are usually not expected to have extensive solution space knowledge. Thus, many of them are unlikely to be able to use the above indicators to identify ASRs. Therefore we need characteristics that are familiar to requirements engineers. We discovered a set of such characteristics. We call them heuristic characteristics. These heuristic characteristics include quality attributes, core features, constraints, and application environment.

3.4.1 Quality Attributes

When a requirement specifies the quality attributes of a software system, it is usually architecturally significant. The participants mentioned a variety of quality attributes from the software systems that they worked on: adaptability, availability, configurability, flexibility, interoperability, performance, reliability, responsiveness, recoverability, scalability, stability, security, extensibility, modularity, portability, reusability, testability, auditability, maintainability, manageability, sustainability, supportability, and usability.

Besides these standard quality attributes, participants also mentioned some particular, project specific quality attributes that are only meaningful to their particular system. For example, in one participant's project, "bettability" was used to express his betting system's ability in attracting users to put bets.

Another observation was that the specific meaning of the quality attributes can vary from project to project. This suggests the need to gather details of quality attributes. For example, only generally stating that, for example, "the system should respond in real-time", is usually not enough to make correct architectural decisions.

3.4.2 Core Features

If a requirement refers to core features of a software system, it is likely to be architecturally significant. The core features define the problems the software is trying to solve. They usually capture the essence of the software system’s behaviour and describe the core expectations users have on the software system. They directly serve to achieve the objective of building the system.

These requirements are usually assumed to be the invariants of the software system, often implicitly. They are part of the fundamental assumptions that the architecture is built upon. Usually, based on the core features of the software system, the most relevant quality attributes choose themselves, as one participant reported: "Based on the unique functional aspects a software system is targeted to address, the non-functional ones often align themselves ... ... A high frequency quantitative trading engine with millisecond latency will automatically emphasize on performance where as an airline control system will immediately focus on reliability."

3.4.3 Constraints

Requirements that impose constraints on a software system are usually architecturally significant. These constraints can be non-technical in nature, such as financial constraints, time constraints, and developer skill constraints. The constraints can also be technical, such as constraints imposed by existing architectural decisions or by technical decisions that have been dictated by a client.

3.4.4 Application Environment

Requirements that define the environment in which the software system will run are likely to be architecturally significant. Some examples mentioned by participants are application servers, the internet, corporate networks, embedded hardware, virtual machines, mobile devices, desktop, laptop, mid-range server, mainframe, operating system, location of servers, etc. Systems running in different environments often have vastly different architectures.

4. Implication on Practice

In this article, we made a distinction between requirements that have significant impact on software architecture and other requirements. This distinction has helped us focus on ASRs when dealing with interplay between requirements and architecture.

ASRs can be challenging to deal with, as revealed by our descriptive characteristics. Descriptive characteristics explain why ASRs are challenging to handle. They can also be useful input for developing evaluation criteria for approaches, tools, and practices to deal with ASRs.

In practice, there is no label for each requirement to tell us whether or not it is architecturally significant. In order to make the distinction practical, we need pragmatic characteristics to enable us to identify ASRs. Indicative characteristics provide such pragmatic hints about the architectural significance of a requirement. They rely on some knowledge of the solution space.

Heuristic characteristics highlight requirements that tend to be architecturally significant, using terminology familiar to people in the problem space. In contrast to indicators, they can be used by people who do not have sufficient knowledge of solution space. They can guide requirements engineers to ask questions proactively, on concerns that are likely to be architecturally significant but that users do not mention. These characteristics have the potential to bring substantial improvements to the gathering of ASRs by identifying those ASRs that might otherwise be ignored.

The twin peaks model suggests that requirements and architecture should be treated iteratively and concurrently. Our observation of the situational nature of ASRs suggests that such iteration is inevitable. For some requirements, only when they are required by architects to make architectural decisions, are they realized as architecturally significant. At this point, architects usually need to ask requirements engineers to provide them.

The findings also reveal another interaction between requirements and architectures: providing feedback on the impact of an architecture on requirements, to inform requirements decisions and to avoid committing to infeasible requirements or requirements that cost more than the value they can generate.

The concrete manifestation of this finding can be in the form of improved requirements negotiation. For example, if a requirement targets a tradeoff point, but does not bring sufficient value to justify the cost incurred by the corresponding design option, one can negotiate the requirement with the users to make the requirement, say, less demanding (or drop it altogether).

This also poses challenges to what used to be traditional thinking that one should not consider solution space concerns while gathering requirements. This can be true for requirements that are not architecturally significant. However, for ASRs, architects' feedback about requirements' impact can help make better informed requirements decisions. Thus, we suggest that appropriate use of solution space knowledge can help gather and negotiate ASRs more effectively.

Indicative characteristics also suggest that there should be a closer collaboration between requirements engineers and software architects. Development teams, processes, and practices should be organized and designed in way that encourages and facilitate such collaboration, with corresponding tools that encourage exchange rather than separation of development activities.

Finally, and perhaps controversially, it does appear that there is a case to be made for requirements engineers have better knowledge of software architectural concerns. We suggest that such knowledge could enable them to gather and negotiate ASRs more effectively. They can ask questions of users more authoritatively during requirements elicitation. However, we are not arguing that solutions could determine problem analysis, but rather inform it and ensure that implications of requirements on architecture (and vice versa) are understood and communicated [10].

Acknowledgements

We thank the participants in our study for their contributions, and the reviewers of our paper for their prompt and thoughtful comments. This work was funded, in part, by the SFI grant 10/CE/I1855 and ERC Advanced Grant 291652.

References

1. Nuseibeh, B., *Weaving Together Requirements and Architectures.* Computer, 2001. **34**(3): p. 115-117.
2. Taylor, R.N., N. Medvidovic, and E.M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. 2009: Wiley.
3. Hofmeister, C., et al., *A general model of software architecture design derived from five industrial approaches.* Journal of Systems and Software, 2007. **80**(1): p. 106-126.
4. Clements, P. and L. Bass, *Relating Business Goals to Architecturally Significant Requirements for Software Systems*. 2010, Carnegie Mellon SEI: Pittsburgh.
5. Glaser, B. and A. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. 1967, Chicago: Aldine Transaction.
6. Adolph, S., W. Hall, and P. Kruchten, *Using grounded theory to study the experience of software development.* Empirical Software Engineering, 2011. **16**(4): p. 487-513.
7. Corbin, J. and A. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. 3 ed. 2007: Sage Publications.
8. Babar, M.A., L. Chen, and F. Shull, Managing Variability in Software Product Lines. Software, IEEE, 2010. **27**(3): p. 89-91, 94.
9. Jorgensen, M. and M. Shepperd, *A Systematic Review of Software Development Cost Estimation Studies.* Software Engineering, IEEE Transactions on, 2007. **33**(1): p. 33-53.
10. Rapanotti, L., et al. *Architecture-driven problem decomposition*. in *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*. 2004.