Towards a Methodology Driven by Relationships of Quality Attributes for QoS-based Analysis*

Steffen Becker
University of Paderborn
Paderborn, Germany
steffen.becker@upb.de

Lucia Happe (Kapova)
Karlsruhe Insitute of Technology
Karlsruhe, Germany
kapova@ipd.uka.de

Raffaela Mirandola
Politecnico di Milano
Milano, Italy
mirandola@elet.polimi.it

Catia Trubiani
University of L'Aquila
L'Aquila, Italy
catia.trubiani@univaq.it

ABSTRACT

Engineering high quality software is a tough task. In order to know whether a certain quality attribute has been achieved or degraded, it has to be quantified by analysis or measured. However, determining what to quantify and how these quantities are related to each other is the difficult part. Early analysis of the quality attributes of a software system on the basis of the system's planned architecture allows informed decisions on design trade-offs. Such decisions can be later validated by measurements on the running system.

In this paper, we revisit software quality attributes. In particular, we introduce a generic taxonomy of quality attributes, the relationship between the attributes is argued, and finally we devise future work leading to an attribute-based methodology for evaluating software architectures. The goal is reasoning about multiple quality attributes of software systems to achieve the ability to quantitatively evaluate and trade-off them.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

QoS-based analysis, quality attributes, relationships.

1. INTRODUCTION

The quality attributes of a software system and their relationships must be carefully understood early in the development process, thus the architect can design an accurate architecture that satisfies them. However, dealing with quality attributes is not a straightforward task. They are usually complex with several relationships and interactions among them, therefore, it is even difficult to make a systematic list of the concerns to be managed. Additionally, some quality attributes have major implications on the core functionality of the system applications and priorities can be useful to roughly fulfill the more relevant end user expectations.

The definition of quality attributes is the first step for defining and enforcing system quality, and the more difficult step is to associate the quality attributes with each other, thus to model their relationships.

In this work, we introduce a generic taxonomy of quality attributes and define relationships between the specified attributes, thus providing easy accessible, early guidance for software architects. The ultimate goal of our work is to enable reasoning about multiple software quality attributes. This would allow to quantitatively evaluate and trade-off quality attributes on the basis of pre-defined relationships.

The paper is organized as follows. Section 2 presents the contribution of this paper, i.e. a graph of relationships to deal with the quality attributes of software architectures and their relationships. Section 3 discusses the benefits of using the graph of relationships as a support to reason and analyze different and conflicting design decisions. In Section 4 we briefly survey the state-of-the-art on the field, and Section 5 concludes the paper and outlines future research directions.

2. GRAPH OF RELATIONSHIPS

Figure 1 illustrates our current proposal of the graph of relationships for quality attributes: nodes (see Section 2.1) represent the quality attributes that we consider, such as *performance*, *reliability*, etc., whereas arcs (see Section 2.2) represent their underlying relationships, such as *trade-off*, *include*, etc.

2.1 Definition of nodes

In this section we provide a brief overview of the quality attributes our graph considers. For sake of space we do not provide a complete description of each quality attribute since it is out of interest of this paper and can be easily retrieved from the international standard for the evaluation of software quality, *ISO/IEC 25010* [1].

*This work has been partially supported by VISION ERC project (ERC-240555).

ICPE'13, April 21–24, 2013, Prague, Czech Republic.

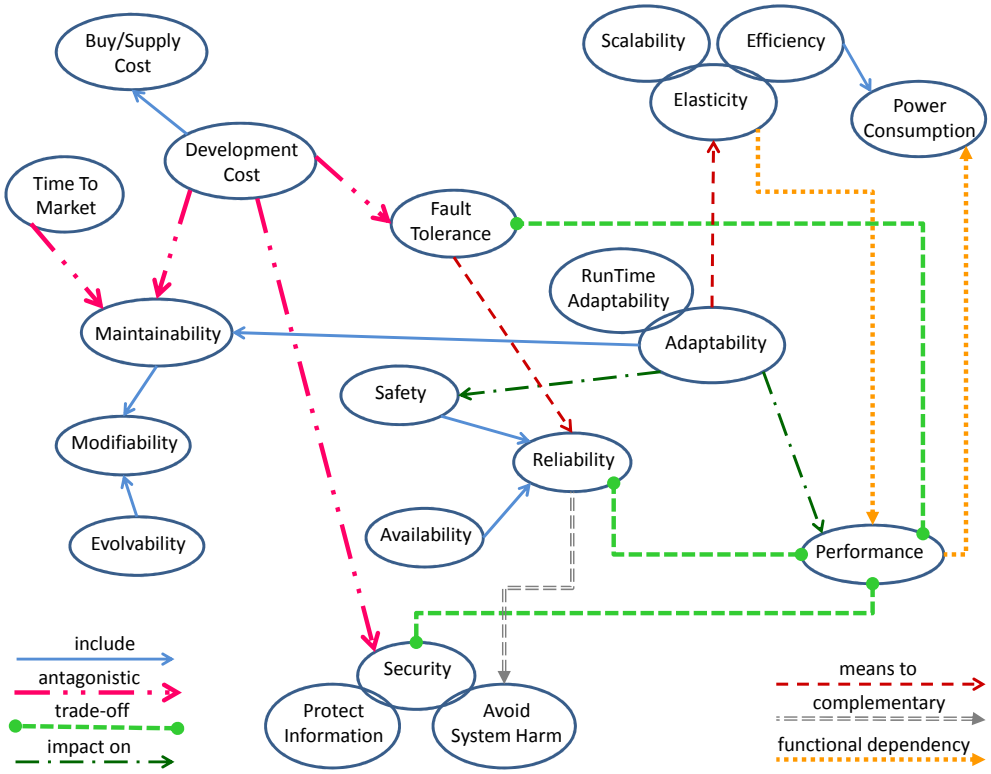

Figure 1: Graph of relationships between quality attributes.

Our graph does not claim to be exhaustive, however, we think it represents a good approximation to conduct a QoS-based analysis. The most common quality attributes are: *performance*, *reliability*, *fault tolerance*, *safety*, *availability*, and *power consumption*. We additionally select other quality attributes that aim at measuring the capacity: (i) to adapt to system changes (*maintainability*, *modifiability*, *evolvability*, *adaptability*, and *run time adaptability*); (ii) to comply with the market constraints (*time to market*, *development costs*, *buy/supply costs*); (iii) to cope with an increasing and/or decreasing number of requests (*scalability*, *elasticity*, *efficiency*); (iv) to protect the system against malicious users (*security*, *protect information*, *avoid system harm*).

2.2 Definition of arcs

In this section we provide a brief overview of the relationships between the quality attributes our graph considers:

include: a quality attribute QA_x includes a quality attribute QA_y if the latter attribute is part of the former one. For example, the *Development Cost includes Buy/Supply Cost* since the latter attribute (i.e. buy software by third party organizations, supply software by implementing system functionalities) is part of the *Development Cost* attribute whose meaning is broader since it contains additional costs (i.e. integration costs, testing costs, etc.).

antagonistic: a quality attribute QA_x is antagonistic with a quality attribute QA_y whether QA_y is negatively influenced by QA_x. For example, the *Time To Market* is *antagonistic* with the *Maintainability* since the property of maintaining a software system is negatively affected by the time it needs to be delivered.

trade-off: it is a bidirectional relationship and it means that two quality attributes (e.g. QA_x and QA_y) depend on each other, in particular the raising of one quality attribute positively or negatively affects the other one. For example, the *Security* and the *Performance* are in a *trade-off* relationship since raising the level of security inevitably affects the performance of the software system.

impact on: a quality attribute QA_x *impacts on* a quality attribute QA_y if the latter attribute is affected by the former one. For example, *Adaptability impacts Performance* since the capacity of the system to adapt on the basis of changes implies that the performance of the system is affected by the actual modifications.

means to: a quality attribute QA_x is a *mean to* a quality attribute QA_y if the latter attribute is achieved through the former one. For example, *Fault Tolerance* is a *mean to Reliability* since fault tolerance mechanisms are aimed at strengthening the system reliability. Several fault tolerance mechanisms can be devised to this scope such as redundancy (i.e. providing multiple instances of the same system to switch in case of failure) or diversity (i.e. providing multiple implementations of the same specification to reduce errors).

complementary: a quality attribute QA_x is *complementary* to a quality attribute QA_y if the latter attribute is indirectly supported by the former one, even if the quality attributes apparently do not seem to be related each other. For example, *Reliability* is *complementary* to *Avoid System Harm* since reliability mechanisms support the avoidance of malicious users by decreasing their probability of intrusion.

functional dependency: a quality attribute QA_x has a *functional dependency* with an attribute QA_y if the latter attribute is used from a functional point of view to cope with the former one. For example, *Performance* has a *functional dependency* to *Power Consumption* since the use of more computing power also requires more electrical power.

3. HOW TO USE THE GRAPH

In this section we discuss the benefits of using the graph of relationships for QoS-based evaluation of software architectures. We demonstrate that the graph can be used as a reasoning (see Section 3.1) and analysis (see Section 3.2) support to guide design decisions of software architects.

3.1 Reasoning on selected quality attribute relationships

In this section we explain the adopted criteria to define the relationships between quality attributes, using our previous experience gained in the field.

Performance vs Security: *trade-off.* The critical aspect that we found between performance and security is the need to quantify the performance degradation incurred to achieve the security requirements. From our previous work [8, 9, 17] we experimented that the solution of a performance model that embeds security aspects allows to quantify the *trade-off* between security and performance in software architectures. In particular, the values of indices coming from the solution of the performance model (i.e. the one that includes security aspects) can be compared to the ones obtained for the same model (i) without security solutions, (ii) with different security mechanisms and (iii) with different implementations of the same security mechanism. Such comparisons help software architects to decide whether it is feasible to introduce/modify/remove security strategies on the basis of (possibly new) performance requirements.

Performance vs Power Consumption: *functional dependency.* The *green computing* discipline addresses the problem of building and managing computing infrastructures that provide the same quality of service with lower energy consumption [16]. Rudimentary techniques for power management, e.g., shutting down idle servers, can impact the ability of the hosting center to meet the Service Level Agreements (SLAs) implicitly or explicitly stipulated with clients. Shutting down servers ensures the maximum power saving; however, bringing up a machine when needed can require a significant start-up time (up to several minutes, depending also on the time needed to start the applications). Start-up delays can severely impact the ability of the system to promptly handle workload fluctuations. Besides, repeated on-off cycles can stress hardware components, increasing the probability of failures, which add further costs for repair or replacement of broken devices.

A different approach suggests to reduce the power consumption by making the system *power proportional* [4], meaning that the power consumption of devices is kept proportional to their utilization. Ideally, a device should consume zero power when its utilization is zero, and full power when its utilization is one. Unfortunately, such "perfect" power proportionality has not been achieved yet.

Fault-Tolerance vs Reliability: *means to.* Reliability can be defined as the probability that a given component or system will perform its required functions without failure, for a given period of time, in a specified environment. When a reliability problem is encountered it means that the service delivered deviates from the correct service, and this is called *service failure*. An *error* is the part of the system state that leads to the occurrence of a failure, and is caused by a *fault*. Therefore, if the system includes mechanisms able to detect and repair errors and faults, the failure probability decreases [2, 18, 14, 15].

There are two well-known strategies for software fault tolerance: error processing and fault treatment [2, 18]. *Error processing* aims to remove errors from the software state and can be implemented by substituting an error-free state in place of the erroneous state, called error recovery, or by compensating for the error by providing redundancy, called error compensation. The second strategy, fault treatment, aims to prevent activation of faults and so action is taken before the error occurs. The two steps in this strategy are fault diagnosis and fault passivation. The nature of faults which typically occur in software has to be thoroughly understood in order to apply these strategies effectively. Techniques for tolerating faults in software have been divided into three classes - design diversity, data diversity, and environment diversity (see [2, 18, 15] for details).

Specifically, these techniques can be integrated in a rule-based approach like proposed in [15]. Once the source of the failure has been identified, through a *sensitivity* analysis for example, a set of rules for the mitigation of the reliability problem, based on fault-tolerance techniques has been defined.

3.2 Further analysis

In this section we discuss the main fields that may benefit from our graph of relationships as a more sophisticated instrument to quickly highlight the quality attributes relationships.

Requirements Analysis. The analysis of the requirements is fundamental since they are aimed at determining the goals, the functions, and the constraints of the software systems. Additionally, multiple stakeholders may define a different priority for the requirements, hence the quality attributes can be associated to stakeholders' utilities/weights. Such analysis can be supported by our graph, in fact it is possible to determine if system requirements are unfeasible depending on the stated relationships between the quality attributes, and the challenge becomes to slightly modify them in order to find the better feasible solution.

Attribute-based methodology for evaluating software architectures. The QoS-based evaluation of software architectures can be performed by looking at their required quality attributes and associating appropriate architectural styles [13], since they represent engineering artifacts defining classes of designs along with their known properties. Architectural styles can be explicitly associated to our graph of relationships that is meant to give suggestions about the most appropriate styles.

Service Level Agreement (SLA) negotiation. A service-level agreement is a part of a service contract where the level of service is negotiated between two parties, i.e. the customer and the service provider. We are interested in SLAs related to the QoS properties of the software systems. For example, these SLAs have SLOs such as the response time, throughput, or utilization for performance, or mean time between failures, mean time to repair for reliability. Our graph of relationships may drive the SLA negotiation by suggesting to the customers the feasible agreements and the ones that may not be fulfilled due to the defined relationships while analyzing the different quality attributes.

4. RELATED WORK

In recent decades, several efforts have been made in the literature for dealing with quantifying software qualities at an appropriate abstraction level. To this end, often software architecture has emerged as a good starting point [7, 19] and several results have been obtained, concerning the definition of methods and tools able to evaluate quality at the software architecture level (see, for example, [3, 10, 19]).

In general, there are *single attribute approaches* that aim at evaluating one quality attribute at time. On the contrary, similarly to the analytic hierarchy process (AHP), there are *multiple attributes approaches* that consider more than quality attribute and mostly aim at comparing the attributes while minimizing and/or maximizing them while looking at optimization techniques (e.g. the Pareto front) that compare two quality attributes at time.

More common today are still methods which address single quality attribute (e.g., performance or availability). However, a major challenge in system development is finding the best balance between different (possibly) conflicting quality requirements that a system has to meet (e.g., maximize performance, maximize availability and minimize cost).

There are multi-attribute approaches where the definition of a software architecture model can embody not only the software qualities of the resulting system, but also the trade-offs decisions taken by designers [5, 6, 20]. Efforts to explore such trade-offs have produced the so-called *scenario-based* analysis methods, such as SAAM and ATAM [11, 12] and others reviewed in [10]. However, these methods provide qualitative results and are mainly based on the experience and the skill of designers and on the collaboration with different stakeholders.

With respect to the state of the art, in this paper we propose a high-level, easy to use, simultaneous analysis of multiple quality attributes with the aim of pointing out their relationships, thus to anticipate negative and/or positive consequences between them. In this way, the software architect has a deeper knowledge of the system under study and architectural decisions can be driven by our graph of relationships.

5. CONCLUSION

This paper presents work in progress towards a systematic approach to do quantitative trade-offs of a whole bunch of quantifiable quality attributes. As a first step, a graph of relationships highlighting the dependencies among quality attributes is presented. This early graph already helps requirements engineers and software architects to check, that not quality attribute has been forgotten. Additionally, trade-offs among conflicting attributes can be identified easily.

In future work, the graph needs to be tested and extended according to experiences. Ideally, quantitative analysis methods are extended to take these relationships into account in order to find systematic and rational trade-offs.

6. REFERENCES

[1] ISO/IEC 25010 - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models, 2011.

[2] A. Avizienis. Fault-tolerant computing-progress, problems and prospects. In *IFIP Congress*, pages 405–420, 1977.

[3] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Software Eng.*, 30(5):295–310, 2004.

[4] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007.

[5] L. J. Bass, F. Bachmann, and M. Klein. Making variability decisions during architecture design. In *Software Product-Family Engineering (PFE)*, pages 454–465, 2003.

[6] P. C. Clements. On the importance of product line scope. In *Software Product-Family Engineering(PFE)*, pages 70–78, 2001.

[7] P. C. Clements. Process validation, session report. In *Software Product-Family Engineering(PFE)*, pages 388–389, 2001.

[8] V. Cortellessa and C. Trubiani. Towards a library of composable models to estimate the performance of security solutions. In *WOSP*, pages 145–156, 2008.

[9] V. Cortellessa, C. Trubiani, L. Mostarda, and N. Dulay. An Architectural Framework for Analyzing Tradeoffs between Software Security and Performance. In *ISARCS*, pages 1–18, 2010.

[10] L. Dobrica and E. Niemelä. A survey on software architecture analysis methods. *IEEE Trans. Software Eng.*, 28(7):638–653, 2002.

[11] R. Kazman, L. J. Bass, M. Webb, and G. D. Abowd. Saam: A method for analyzing the properties of software architectures. In *International Conference on Software Engineering (ICSE)*, pages 81–90, 1994.

[12] R. Kazman, M. H. Klein, M. Barbacci, T. A. Longstaff, H. F. Lipson, and S. J. Carrière. The architecture tradeoff analysis method. In *ICECCS*, pages 68–78, 1998.

[13] M. H. Klein, R. Kazman, L. J. Bass, S. J. Carrière, M. Barbacci, and H. F. Lipson. Attribute-based architecture styles. In *WICSA*, pages 225–244, 1999.

[14] J.-C. Laprie. Dependability modelling and evaluation of software and hardware systems. In *Fehlertolerierende Rechensysteme*, pages 202–215. Springer, 1984.

[15] Q-ImPrESS Consortium. The Q-ImPrESS project. Project website: http://www.q-impress.eu, 2010.

[16] P. Ranganathan. Recipe for efficiency: principles of power-aware computing. *Commun. ACM*, 53:60–67, April 2010.

[17] R. J. Rodríguez, C. Trubiani, and J. Merseguer. Fault-Tolerant Techniques and Security Mechanisms for Model-based Performance Prediction of Critical Systems. In *ISARCS*, 2012.

[18] K. S. Trivedi. Reliability evaluation for fault-tolerant systems. In *Computer Performance and Reliability*, pages 403–416, 1983.

[19] L. G. Williams and C. U. Smith. Pasa$^{\text{sm}}$: a method for the performance assessment of software architectures. In *WOSP*, pages 179–188, 2002.

[20] J. Yang, G. Huang, W. Zhu, X. Cui, and H. Mei. Quality attribute tradeoff through adaptive architectures at runtime. *Journal of Systems and Software*, 82(2):319–332, 2009.