# Considering Not-quantified Quality Attributes in an Automated Design Space Exploration

Axel Busch*, Anne Koziolek*
*Karlsruhe Institute of Technology, Germany
{busch, koziolek}@kit.edu

***Abstract*—In a software design process, the quality of the resulting software system is highly driven by the quality of its software architecture (SA). In such a process trade-off decisions must be made between multiple quality attributes (QAs), such as performance or security, that are often competing. Several approaches exist to improve SAs either quantitatively or qualitatively. The first group of approaches requires to quantify each single QA to be considered in the design process, while the latter group of approaches are often fully manual processes. However, time and cost constraints often make it impossible to either quantify all relevant QAs or manually evaluate candidate architectures. Our approach to the problem is to quantify several most important quality requirements, combine them with several not-quantified QAs and use them together in an automated design space exploration process. As our basis, we used the PerOpteryx design space exploration approach, which requires quantified measures for its optimization engine, and extended it in order to combine them with not-quantified QAs. By this, our approach allows optimizing the design space by considering even QAs that can not be quantified due to cost constraints or lack of quantification methodologies. We applied our approach to two case studies to demonstrate its benefits. We showed how performance can be balanced against not-quantified QAs, such as security, using an example derived from an industry case study.**

***Index Terms*—Not-Quantified, Requirements, Design, Decision, Architecture, Trade-off, Automated**

## I. Introduction

Today's software systems increasingly have become larger and more complex than in past years. More stakeholders are involved, more complex tasks are processed by software systems and thus, responsibility is delegated to software systems. Thus, the developer teams have become larger. The CBSE paradigm proposes to divide large projects in smaller subsystems that can be developed separately. This helps to cope with the size of the project, but at the same time increases the complexity of the software architecture (SA). The requirements specification of such a software project contains lists of functional and quality requirements that need to be considered.

Transforming a requirements specification in a SA is one major part of developing software systems. The process of creating an architecture design typically involves making architecture design decisions to consider the requirements of a system. Making design decisions does not solely result in functionality, but highly influences the resulting quality and therefore the quality attributes (QAs), such as performance, of a SA. While functional requirements are often developed as a complement, QAs are often contradictory in nature. For instance higher performance often implies higher costs, or reduces maintainability. Thus, we need a methodical approach to systematically analyze the impact of design decisions on the QAs of a SA to allow architecture trade-off decisions, even if we have no adequate quantification methodology.

But making architecture trade-off decisions according to the requirements is not trivial. The huge space of architecture candidates makes it difficult to manually find the optimal solution for the considered requirements even for small SAs. This becomes important when software architects have to think about the performance implications when adding a new security feature to the system. From this single requirement, a huge number of architecture candidates arise—all coming with different resulting performance. Assessing them and making trade-off decisions are time-consuming and error-prone when evaluating this by hand and requires experience of the architects. In recent years, several approaches shifted the traditional experience-driven craftsmanship process of architecture design towards a systematic engineering process. Becker et al. provide a component model incorporating the advantages of formalized architecture description languages, i.e. UML, to get a formalized representation of an architecture at hand [1]. Moreover, it provides simulation engines to predict QAs at design time. On top of this, the PerOpteryx approach [2] allows automated design space exploration resulting in Pareto-optimal solutions for quantifiable QAs, namely as performance, reliability, and costs. The approach reduces the complexity of making architecture trade-off decisions when considering several QAs. Several other approaches improve a given architecture considering QAs, such as ArcheOpteryx [3], and ArchE [4].

However, quantifying each QA that has to be considered is not possible in many cases. One reason is the high effort of quantifying QAs that is often very time-consuming and therefore may exceed the project budget. Furthermore, some QAs, such as security, lack an applicable methodology to get adequate measures in a manner to be used in such an automated design space optimization approach. Often, however, these kind of quality attributes of software components can at least be estimated and incorporated in this way in the exploration. But since no approach allows to consider the aforementioned not-quantified QAs, they often remain unconsidered when optimizing automatically, even they are main requirements of the project.

The contribution of this paper is an approach to consider not-quantified QAs of software architectures in an automated design space exploration in order to give an architect feedback

when making architecture trade-off decisions. We extended the PerOpteryx approach to consider them in its automated design space optimization engine. PerOpteryx results in Pareto-optimal architecture solutions that can be used to select suitable architecture decisions meeting the project requirements in consideration of not-quantified QAs best. We developed a metamodel to consider not-quantified QAs together with quantified QAs. We included our metamodel into PerOpteryx and built an extension to compute the not-quantified QAs in its design space optimization engine. Finally, we applied our approach on two case studies to demonstrate its benefits on an industry-related system.

This paper is organized as follows: Section II further motivates our methodology. Section III provides a general overview of the architecture model Palladio, the design space optimization approach PerOpteryx, which we use as a basis for our methodology, and the Quality of Service Modeling Language, which we use to define our not-quantified QAs. Section IV shows an overview and describes our methodology. Section V presents our models to represent and evaluate not-quantified QAs. In Section VI, we demonstrate our methodology in two case studies. Finally, Section VII discusses related work, while Section VIII concludes the paper and discusses future work.

## II. Motivating Scenario

In a SA decision making process, a software architect is typically interested in the effects of the decisions on the QAs of the software architecture. Such decisions are typically taken to improve a QA (for example a decision to add a database abstraction layer to improve maintainability), but also decisions derived from functional requirements (for example a decision to create an interface to an additional external payment method in a business application). However, at the same time, these decisions also affect additional QAs, such as performance.

Adding an external payment method may have, for instance, impact on the performance, the usability, as well as the security of a system. The performance may be influenced at least due to the response time of an external service. The system response time could be lower, due to less load on the own server, but could also be higher due to an overloaded external service. In extreme cases, the own service could be unreachable in case of an external service that is offline.

Further, the usability is influenced, for instance, due to a changed workflow a user needs to undertake to finish his purchase. Similar as in the previous case, the usability may be improved due to less clicks or worsen due to more clicks and a less intuitive workflow.

To balance against different QAs a software architect can use a design space optimization approach to find the best solution according to the project requirements. To do so, the architect could start to create a performance model, and analyze the usability of the graphical user interface (GUI).

However, creating detailed quality models can be very time-consuming. Especially in case of security quality it may be hard to set up such a detailed model. Due to these time-consuming modeling processes, architects may tend to omit the process of systematically consider all relevant QAs of the system. In our approach, architects can choose which QAs to quantify and which ones to consider in a more lightweight fashion based way on rough estimations (which we call modeling *not-quantified QAs*). At the same time, they can consider both the modeled quantified, and not-quantified QAs in a single automated design space exploration.

## III. Background

This section introduces background on the modeling and prediction of quantified QAs that we build our approach on. In Section III-A, we describe the Palladio Component Model (PCM), a SA modeling language. For our approach, we use PCM as our component model. Section III-B describes PerOpteryx, an approach for automated design space exploration considering quantified QAs. In this paper, we use PerOpteryx as a basis for our approach and extended it in order to additionally consider not-quantified QAs in the design space optimization. Finally, we show in Section III-C the basic concepts of the quality of service modeling language (QML) [5], which we use to define quality dimensions and exploration objectives.

### A. Architecture Model: Palladio Component Model

The Palladio Component Model (PCM, [6]) is a metamodel for component-based SAs and also provides a set of analysis tools for performance, reliability, and costs evaluation. This introductory section is taken from [7].

The PCM is specifically designed for component-based systems and strictly separates parametrized component performance models from the composition models and resource models. Thus, the PCM naturally supports many architectural degrees of freedom (e.g., substituting components, changing component allocation, etc.) for enabling automated design space exploration.

Consider the minimal PCM model example in Fig. 1, which is realized using the Ecore-based PCM metamodel and visualized here in UML-like diagrams for quick comprehension. The architecture model specified by the software architect consists of three connected software components `BusinessTripMgmt`, `BookingSystem`, and the internal `PaymentSystem` deployed on three different hardware nodes. The software components contain cost annotations, while the hardware nodes contain annotations for performance (processing rates) and cost (fixed and variable cost in an abstract cost unit).

The example system used here is a Business Trip Management system with booking functionality. Users are administrative employees that plan and book journeys.

For each software component service, the component developers provide an abstract behavioral description called service effect specification (SEFF). SEFFs model the abstract control flow through a component service in terms of internal actions (i.e., resource demands accessing the underlying hardware) and external calls (i.e., accessing connected components). Modeling each component behavior with separate SEFFs enables us to quickly exchange component specifications without the need to manually change system-wide behavior specifications (as required in e.g. UML sequence diagrams).

For performance annotations, component developers can use the extended resource-demanding service effect specifications

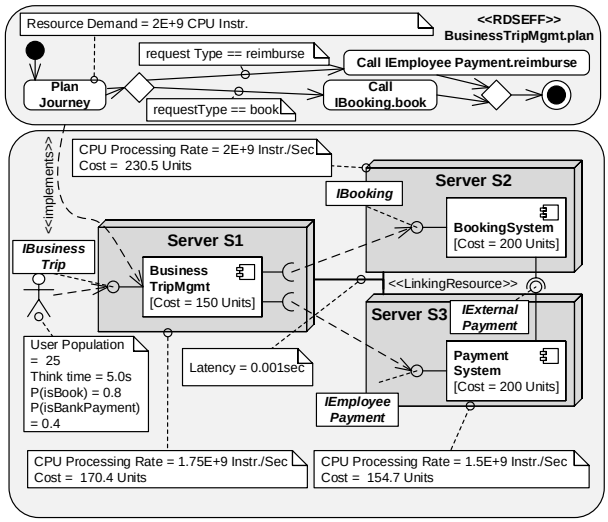


Fig. 1: Example PCM Model: Business Trip Booking System (RDSEFFs). Using RDSEFFs, developers specify resource demands for their components (e.g., in terms of CPU instructions to be executed). An example RDSEFF is shown at the top of Fig. 1 for the service `BusinessTripMgmt.plan`.

A software architect composes component specifications by various component developers to an application model. The performance simulation finally is performed by solving such a model analytically or simulation-based. In this paper, we use the transformation from PCM to Layered Queueing Networks [8].

### B. Design Space Exploration: PerOpteryx

We apply our methodology based on PerOpteryx [9], but the concepts are not limited to this approach. The PerOpteryx approach [7] explores the design space of a given SA and thus supports to make well-informed trade-off decisions for performance, reliability, and costs. This introductory section is taken from [7].

For the exploration, PerOpteryx makes use of so-called *degrees of freedom* of the SA that can either be predefined and derived automatically from the architecture model or be modeled manually by the architect. In our example (Figure 1), we have two types of predefined degrees of freedom: We can change the component allocation and the server configuration. For the allocation degree, we could have several servers, each having different possible processing rates. As an example of a manually-modeled degree of freedom, let us consider that some of the architecture's components offer standard functionality for which other implementations (i.e. other components) are available. In this example, let us assume there is a fourth available component `QuickBooking` that can replace `BookingSystem`. Assuming that `QuickBooking` has less resource demand but is also more expensive than `BookingSystem`, the resulting architecture model has a lower response time but higher costs.

The resulting degree of freedom instances (DoFI), which span a design space which can be explored automatically. Together, they define a set of possible architecture models. Each of these possible architecture models is defined by choosing one design option for each DoFI. We call such a possible architecture model a *candidate model*. The set of all possible candidate models corresponds to the set of all possible combinations of the design options. We call this set of possible architecture models the *design space*.

Using the quantitative quality evaluation provided by the PCM analysis tools, PerOpteryx can determine performance, reliability, and cost metrics for each candidate model. In general, to quantify a QA $q$, we choose a quality metric $m(q)$. The quality evaluation $\Phi_q$ for a QA $q$ can be expressed as a *quality evaluation function* from the set of valid PCM instances $M$ to the set of possible values of the quality metric $m(q)$, denoted $\mathcal{V}_{m(q)}$: $\Phi_q : M \rightarrow \mathcal{V}_{m(q)}$. In addition to the evaluation functions, PerOpteryx requires a specification whether a quality is to be maximized or minimized.

Based on the DoFIs (as optimization variables) and the quality evaluation functions (as optimization objectives), PerOpteryx uses genetic algorithms and problem-specific heuristics to approximate the Pareto-front of optimal candidates. Details on the optimization are not required for the discussion in this paper, but can be found in [10], [7].

In its previous version described in this section, PerOpteryx does not support the effect of decisions on other, not-quantified QAs, such as usability or security. The main effects of this decision on usability and security would remain hidden. Thus, such a decision to add an external payment method cannot be meaningfully studied with the previous version of PerOpteryx.

### C. Quality Declaration: QoS Modeling Language

The Quality of Service Modeling Language allows the specification of QAs, quality dimensions, and quality requirements. It was originally defined in Extended Backus-Naur Form [5]. In previous work [11], we have extended it for the use with PerOpteryx and built a metamodel in the Eclipse Modeling Framework (EMF). Additionally, we have simplified the language for this paper (by removing the contract types). A model instance of the resulting language (abbreviated QML in the following) are comprised of three levels: The dimensions, the contract, and the profile.

- *Dimensions:* The dimension declaration (Figure 2) defines relevant quality *dimensions*. These dimensions can be quantified quality measures such as mean response time or probability of failure on demand, but also qualitative QAs such as user satisfaction (as a sub-characteristic of usability) or access restriction capability (as a sub-characteristic of security) that shall be considered by the design space exploration. Dimensions specify their domain of possible values as a set of numbers, a set of strings, or a multiset of strings. Additionally, a dimension $d$ may declare an partial or total order $<_d$ on their elements and whether smaller values or larger values (according to $<_d$) are preferable.
- *Contract:* The contract (not shown) specifies *constraints* and/or *objectives*. A constraint defines a quality requirement. This can be a required quality value, or a lower or upper bound on the quality of an order is available. An objective specifies that a certain dimension shall be improved as much as possible.
- *Profile:* The profile (not shown) assigns a contract to a system interface or to a system service. This interface or service shall fulfill the requirements defined in the

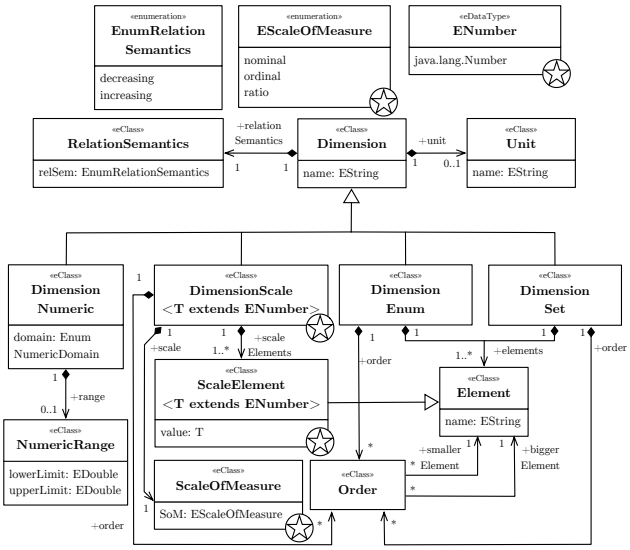


Fig. 2: QML Dimension [11], [5] and metamodel extension (marked with asterisk)

contract. Additionally, the quality dimensions specified as objectives in the contract shall be improved as much as possible for this interface or service.

## IV. Methodology overview

Figure 3 shows an overview of our methodology. It is comprised of three parts: The analysis input models (Figure 3: (1)), the analyzing process (Figure 3: (2)) and the result of the analysis (Figure 3: (3)). We use three input models (1) for our analysis:

- *Quality Annotations Model:* The Quality Annotations Model (QAM) allows to annotate QAs to software architecture components. We distinguish two different kinds of QAs: The quantified and not-quantified QAs. While for the quantified QAs a methodology or metrics exist to quantify them, this is different for the not-quantified attributes. For them, either no quantification methodology exists, or applying the existing solution may be too time-consuming.
- *Software Architecture Model:* The Software Architecture Model represents the SA of the application including its components, hardware context, usage context, and called external services. Since the PCM approach fulfills our requirements, we use this as a basis for our SA model.
- *Quality Declaration Model:* The Quality Declaration Model allows to specify the dimensions of the QAs to be fulfilled by a SA. Bounds can be defined that describe the lower and upper limits of the QAs to be fulfilled by the architecture. Alternatively, quality requirements can be specified as objectives, i.e. defines the requirements to be improved. QML fulfills most of the required entities. Thus, we use QML as a basis to declare our quality requirements and extended it for our purposes.

In the analysis process (2), we use the input models to search for the Pareto-optimal architecture candidates (3) that best fulfill the defined quality requirements by using PerOpteryx. Finally, the software architect selects the best resulting architecture according to the software project's requirements.

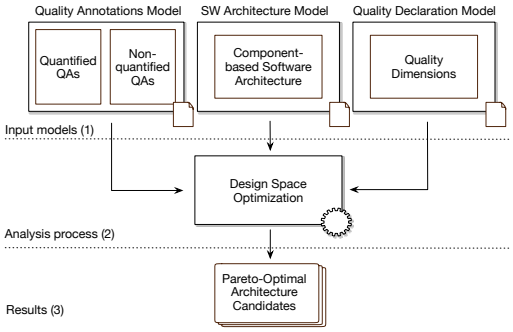


Fig. 3: Quality analysis methodology

## V. Not-quantified QA Model

This section introduces terminology around our extension and our model for not-quantified QAs to consider them in an automated design space exploration together with already quantified attributes. Further, we describe our quality annotations model, and the evaluation function for our candidate evaluation.

### A. Terminology

*1) Quality attributes:* Quality attributes describe the non-functional properties of a SA. According to the requirements on a software project a different set of QAs becomes relevant. *Example:* In the onboard navigation system of an airplane or in other safety critical systems the QA reliability is more likely in the set of considered QAs than in the board entertainment system of the same airplane. As you can see, the set of QAs to be considered is highly influenced by the requirements of the project. We distinguish two kinds of QAs:

- *Quantified quality attributes:* For quantified QAs either a methodology or a metric exist to quantify them and make the attribute tangible. Further, the benefit of applying the QA quantification methodology dominates the required quantification effort.
- *Not-quantified quality attributes:* Not-quantified QAs of a software architecture could have three different reasons: The first reason is they may be unimportant for the project and due to their pettiness they simply remained unconsidered. Second, a methodology or a metric may exist, but because the effort of the quantification dominates the quantification benefit, they remained unconsidered. For example, the Palladio approach provides a methodology to consider reliability of a component-based SA. But due to the time-consuming methodology the architects may decide to leave reliability unconsidered in favor of other, more important quality requirements. Third, there may exist no methodology or metric to quantify them or the existing methodology or metric is not applicable in the considered scenario. An example for the third reason is security. In many cases security is a highly important requirement, but remains unconsidered, since there is no methodology to quantify this attribute.

*2) Requirements:* As described in the previous section, the quality requirements, and especially in a software design process the considered quality requirements are driven by the project requirements.

### B. QML extension

QML as a language to specify QAs and requirements provides most of the entities to define not-quantified QAs. However, it lacks entities for specifying values on different levels of measurement. We extended the QML dimension and the contract metamodel to address this issue.

*1) Dimensions:* Figure 2 shows the metamodel of the QML dimensions (based on [11]). For instance costs of a SA is comprised of several quality `dimensions`: It could be for instance comprised of `initial costs`, `maintenance costs`, or `operation costs`, while the unit could be *monetary units*. Each of these three types of costs can be represented by dimensions. Each `Dimension` specifies the possible values that are possible to be assigned. Continuous numeric values, as well as discrete String values are possible. Costs, would be modeled using `DimensionNumeric`, since costs typically lie in a certain range. Minimum and maximum costs for a concrete architecture can be determined. Further, there is a `RelationSemantic` for each dimension. The `RelationSemantic` specifies if more or less means better. Intuitively, an architect may tend to minimize costs, but maximize security. By using `DimensionEnum` and `DimensionSet` discrete String values can be modeled. As a fourth dimension type, we introduced the dimension scale. By this, elements by one certain type of `ENumber` can be instantiated. The modeler has to decide to instantiate a certain type of elements: dimension scale can be tailored to a set of Integer, Short, Double, Float or Byte, dependent on the values a QA anticipates. It is comprised of several `ScaleElements` that conform to the concrete type of dimension scale. Further, dimension scale specifies a level of measurement (`ScaleOfMeasure`) for its comprising elements. The elements can conform to an ordinal, nominal or ratio level of measurement. By using `Order`, the elements of the enumeration, set, and scale can be ordered by defining smaller and bigger elements.

*2) Contract:* We extended the metamodel of the QML contract to achieve our goals (based on [11]). The contract specifies quality requirements or quality constraints of valid architectures by defining a `Criterion`. A constraint defines an upper or lower bound for a dimension. A valid architecture candidate must fulfill this constraint. An objective specifies a certain dimension to be improved as much as possible. A criterion can have one or more `EvaluationAspects` and, optionally, an `AspectRequirement`. An evaluation aspect is either specified as deterministic (`DeterministicEvaluationAspect`) or stochastic (`StochasticEvaluationAspect`). The latter is either a `Frequency`, that defines a `RangeValue` in which the QA can vary, or by a `PointEstimator`. The aspect requirement has an aspect requirement literal, that corresponds to an `EnumLiteral`, `SetLiteral`, or `NumericLiteral` (for double values) and is defined in the corresponding dimension. As a fourth type, we introduced the `ScaleLiteral`. Scale literal defines a set of numeric values limiting the space of possible values to be assigned to a QA. The scale level conforms to the `ScaleOfMeasure` of the referenced dimension.

*Example:* In our example, we consider performance as quantified QAs of the running example in Section III, and additionally two not-quantified QAs, namely usability and security. We model one dimension for each quality attribute, such as the response time for representing the performance of the system. The response time is represented by a numeric dimension with the unit milliseconds and decreasing relation semantics, i.e. smaller values are better than larger ones. Next, a contract can either constrain a dimension or define an objective for the optimization. For performance, let us assume that we require a mean response time of less than 500 ms for our service. Thus, we define a constraint in our performance contract.

The QAs usability and security are both not-quantified QAs in our example. For usability, we define the dimension `UserSatisfaction`. Different payment system components could fulfill the user satisfaction in a different way. One component could either more satisfy the user, while others are less usable, and therefore would less satisfy the user. We model this with different possible enumeration values $\{\{low\}, \{middle\}, \{high\}\}$ for the user satisfaction dimension. Additionally, we define an order that specified that low is smaller than medium is smaller than high, i.e. $\{low\} < \{middle\} < \{high\}$. Finally, we set the relation semantics to *increasing* to express that user satisfaction shall be maximized. For our contract, we refer to user satisfaction as an objective, which means that this dimension shall be optimized. Note that the implicit underlying scale of the user satisfaction dimension is ordinal as well, as we have defined a total order for it.

For security, we define a dimension `Access-Restriction`, which, for the sake of brevity, shall reflect the major security concern in our example. As before, different implementations of the booking system component may differ in their capabilities of prevent unauthorized access. Thus, we define the access restriction dimension, that represents different levels of security. Let us assume $\{1, 2, 3, 4, 5\}$ are possible values. The order is already implied by the used integer values, so we only need to set the relation semantics to increasing to express that the higher value is better. Additionally, we define that the underlying scale is ordinal, which tells the optimization that operations such as difference and multiplication are not meaningful on this scale.

We define objectives for both, since we want to find the architecture that is as best fitting as possible. Costs are omitted due to the comprehensibility of the graphic, but would be modeled in the same manner.

### C. Quality Annotations model

The quality annotation model annotates a quality value to components in the SA. For example, we can express that our `BookingSystem` component has access restriction quality "high". The concrete quality value of a dimension needs to be estimated by the architect.

Figure 4 shows the metamodel of our Not-quantified QA model (NQ2A). An `NQ2ARepository` holds one or several `NQ2A`. `NQ2A` is the relation between a software component in the PCM (a `PCM::RepositoryComponent`,

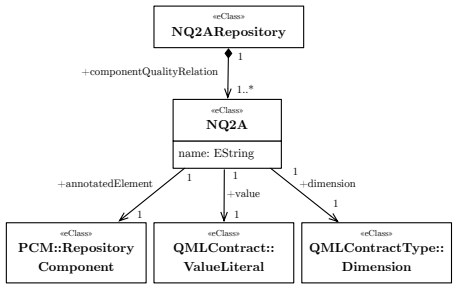


Fig. 4: Not-quantified quality attribute annotation model

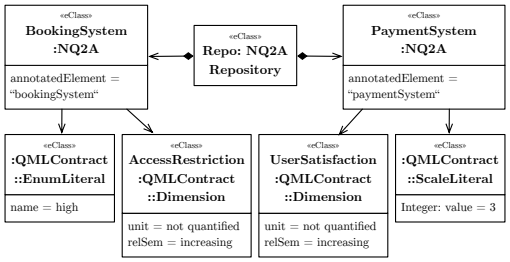


Fig. 5: Instance of not-quantified QA annotation model

such as the `BookingSystem` component) and its quality property (`QMLContract::ValueLiteral`, such as the value "high"). Additionally, the dimension for quality (such as mean response time) is referenced to put the value into context. *Example:* Figure 5 shows an example instance of the NQ2A model. In our running example, we want to express a security assessment of the `bookingSystem` component and a usability assessment of the `paymentSystem` component. Using our NQ2A model, we attach quality properties to components of our system architecture: The `NQ2ARepository` contains two elements of type `NQ2A` holding the quality property and component relations for the two components. Figure 5 shows one concrete instance for each QA. More concrete, the `BookingSystem` NQ2A instance connects the booking system component with the access restriction quality "high". This means the booking system component's access restriction capability is estimated to be "high". For the access component, the `PaymentSystem` NQ2A object attaches the value "3" describing the user satisfaction to the payment system component of our system.

In practice, for an automated exploration of architecture candidates, we could define for instance several components of identical functionality, but different QAs for a degree of freedom that realizes component variation. For instance, another booking system component `bookingSystem'` with different resource demands, and access restriction capability "middle". A system architecture using the `bookingSystem'` component is then an architecture alternative to an architecture using `bookingSystem`. Here, both architectures are functionally equivalent, but may differ in their QAs.

### D. Candidate evaluation

With the quality annotations model defined in the previous section, estimations for the not-quantified quality dimensions can be annotated to components of the architecture model. However, as we are considering not-quantified QAs, no theory of how to derive system-level quality metrics from these annotations is available (unlike for example performance). For not-quantified QAs, no theory of composition from individual component quality to overall system quality is available in general.

To cope with this problem, we consider the quality value of a single component as a separate quality dimension for the design space exploration. Let $d$ be a quality dimension (such as user satisfaction) and let $v_d(m)$ be the quality value annotated to a component in a candidate model $m$ (such as the value 3 in Fig. 5). For each candidate model in the design space exploration, there must be exactly one quality value per not-quantified quality dimension. Then, we define a simple evaluation function $\Phi_d$ from the set of valid PCM instances $M$ to the set of possible values $\mathcal{V}_d$ of the quality dimension $d$ (as defined with the QML Dimension metamodel in Fig. 2) as $\Phi_d : M \to \mathcal{V}_d$ where for a candidate model $m$, $\Phi_d(m)$ is the annotated quality to the annotated component: $\Phi_d(m) = v_d(m)$

With this evaluation function $\Phi_d$ and if an order is defined for a not-quantified quality dimension, the design space exploration can consider this dimension as an objective for the exploration process. If no order or only a partial order is defined, the quality dimension can still be used to define constraints and goals.

To ensure that each candidate model has exactly one quality value per relevant quality dimension, we perform two checks on the input architecture model. First, only one quality value per relevant quality dimension must be annotated to the input architecture model. Second, each component selection degree of freedom that replaces a component with an annotated quality value must provide only design options that have exactly one quality value of the same quality dimension each.

This evaluation function—even though simple—, allows us to consider the architect's reasoning on these not-quantified QAs. We believe it is more beneficial to consider the not-quantified QAs in this simple form than not considering them at all.

A side remark concerning future work: we are planning to add aggregation functions that the software architect can use to express her own assumptions about the relation between individual component quality and overall system quality to capture as much as possible of the architect's reasoning in the exploration.

## VI. Evaluation

We applied our approach on two case studies, to demonstrate the applicability and benefits of our approach. The evaluation aims at demonstrating the benefit when combining quantified QAs together with not-quantified QAs in a decision making process. First, we applied our approach on the Business Reporting System (BRS), a system that is used to retrieve business processes and statistical analyses of these processes. The BRS system is loosely based on a real system [12].

The second case study shows the application of our approach on a real-world system, called the Remote Diagnostic Solutions (RDS).

### A. Experiment overview

In this evaluation, we show how a combination of quantified QAs with not-quantified QAs can help to make trade-off

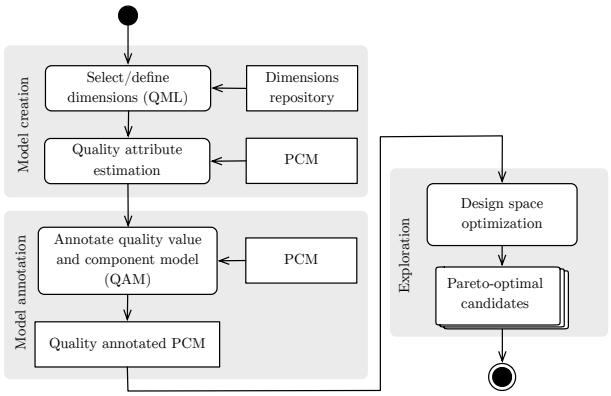


Fig. 6: Evaluation methodology

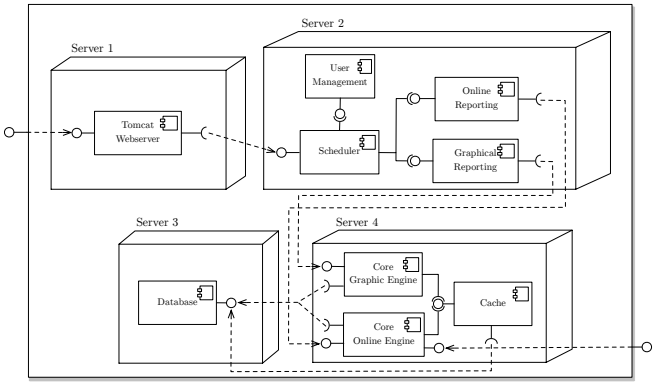


Fig. 7: System architecture of the BRS

decisions between several QAs on software architectures. Thus, our case studies follow a three steps execution plan that is shown in Figure 6: First, we create the models required for the optimization. We select predefined dimensions from a dimensions repository or define dimensions using our extended QML model to be incorporated in the optimization. Further, we decide which dimensions should be quantified and which are treated as not-quantified. For the not-quantified QAs, we estimate the concrete quality values of the dimensions for the concerned component of the PCM. Second, we annotate the previously estimated quality values to the corresponding PCM components using our QAM. In the third step, we execute the design space optimization to calculate the Pareto-optimal candidates. The resulting Pareto-optimal candidates can then be used to make trade-off decisions between the considered QAs.

### *B. Case Study I: Business Reporting System*

We use the business reporting system for our first case study. The BRS allows a user to retrieve business reports and statistical analyses about running business processes from a database. The system is loosely based on a real system [12]. Since the system interacts with the user and stores data it may become important to include usability and security dimensions in the trade-off process.

*1) System Setup:* Figure 7 shows an excerpt of the system architecture combined with the initial hardware topology. The initial BRS configuration is a 4-tier system, that is comprised of a web server (server 1), two servers for the business logic (server 2 and 4), and a database server (server 3).

The user issues a request to the `webserver` component to create a graphical report. These user requests are delegated to the `Scheduler` component, which forwards the request either to the `GraphicalReporting` component, or to the `OnlineReporting` component, depending on the type of request. If the user requested a graphical report, the request is delegated to the `CoreGraphicEngine`. If the plain data was desired the `CoreOnlineEngine` is called. Both components call the `Database` components. To relieve the database server, some requests are served by the `Cache` component. For user management, the `Scheduler` can first call the `UserManagement` component for session handling. For maintenance purposes, the `CoreOnlineEngine` can be accessed directly by a maintainer.

In this case study, we determine the performance, and the costs of the system as quantified QAs. To model the behavioral view (performance) of the system, we use the RDSEFF of Palladio. The performance and cost model are reused from our previous work [2]. To evaluate the performance, we transform the model to Layered Queuing Networks and use the solver by Franks et al. [13].

*2) Experiment execution:* In this experiment, we consider usability and security as two not-quantified QAs in addition to performance and costs (as quantified QAs). By including usability and security in the decision making process, the experiment provides Pareto-optimal candidates of the architecture under all four aspects.

For the experiment, we carry out a component selection scenario. We include several new components to our repository that serve as architecture alternatives. All these alternatives are functionally equivalent, but differ in their quality, which in turn influences the quality of the SA. So first, we added two further components to our repository:

Let us assume that we have a more secure `UserManagement'` component available. Due to the higher security quality, we assume a doubled CPU resource demand by this component.

The second component `GraphicalReporting'` may be an improved implementation of the `GraphicalReporting` component that is more usable. The implementation provides a higher user satisfaction than the previous implementation. We estimate for this component a fourfold CPU resource demand due to the usability improvements.

Both components have identical provided and required interfaces to the original ones. Thus, PerOpteryx uses the alternatives autonomously in the design space exploration run.

For the `UserManagement'` component we specify a doubled CPU resource demand (in the RDSEFF model), and a fourfold demand for the `GraphicalReporting'` component.

In the next step, we create the model estimating the not-quantified QAs for the four components (base components and alternative components) and connect them to the intended components, as described in Section V-C. We create a security dimension `AccessRestriction` that represents the access restriction capability of the target components. We specify three possible values a component can take for the access restriction dimension: *low*, *middle*, and *high*. These three values estimate

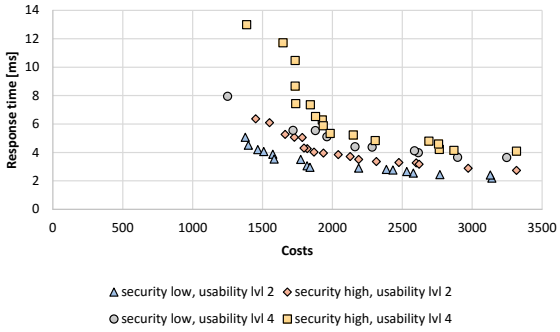


Fig. 8: BRS system: Resulting Pareto-optimal candidates for performance, costs, security, and usability quality

the expected security level on an ordinal level of measurement. Further, we set up a second dimension `UserExperience` representing the usability quality of the SA. For our experiment, we specify five levels resulting in the set $\{1, 2, 3, 4, 5\}$, where a higher value means higher quality.

In the next step, we apply the quality characteristics for each dimension to our architecture model. Further, we estimate and annotate the base `UserManagement`'s component access restriction capability to be *low*, while we annotate *high* to the alternative component `UserManagement'`. In the same manner, we estimate the user experience for the `GraphicalReporting` component to be *2* and for the alternative implementation to be *4*.

This model can now be used in combination with the architecture model to be optimized in PerOpteryx. The analysis is executed using a constant usage profile for the performance simulations. The resulting Pareto-optimal candidates is described in the following section.

*3) Results:* We performed 200 iterations evaluating in total 2586 architecture candidates. Out of these, 63 architecture candidates are Pareto-optimal. The Pareto-optimal results of the optimization are shown in Figure 8.

Since the resource demands of the higher quality components are higher (at constant usage), we expect higher costs and/or higher response times for the candidates implementing these components. In reverse, the original lower quality components should be assembled to the architecture candidates with lower cost and lower response time. Analyzing the Pareto-optimal results confirms our expectation. The candidates with *low* security and level *2* usability are best in performance and costs than all other candidates, while the candidates with *high* security and level *4* usability show worst performance and usability, as expected. The remaining candidates are in the midfield: The *low* security, level *4* usability candidates are more expensive and show higher response times than the *high* security, level *2* usability candidates. This is plausible, because the resource demands have been quadrupled for the usability improvement and only doubled for security improvement (and because the actual resource demands of the two original components were similarly high).

Using these Pareto-optimal results, an architect could now decide which architecture candidate to choose in accordance to the performance, costs, level of security and usability that is necessary for the project. For instance an architect could decide to choose the architecture candidate with security high instead

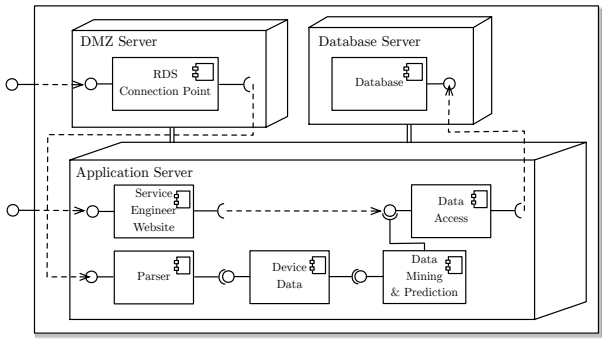


Fig. 9: System architecture of the RDS [14]

of security low, even if this is slightly more expensive, but in combination with the usability level 4 components would be too expensive. Another decision may be to select highest security, highest usability (level 4), and a response time less than 6 ms. Here, the architect could derive that under the given requirements the lower cost barrier would be at 2000 cost units. As we can see, the actually chosen architecture is highly dependent on the demanded requirements and available resources. Our approach makes the trade-offs explicit and thus making the decision easier and more usable.

### C. Case Study II: Remote Diagnostic Solutions

The Remote Diagnostic Solutions (RDS) is an industry-related software system used to record device information, failures and other status information of industrial devices [14]. The RDS offers two different services: First, industrial devices can access the system and upload their diagnostic status information. In case of an abnormal behavior the devices upload further error information that can be analyzed by the operators. Their service engineers can then access the system by a website allowing to generate reports about the status of the devices. Additionally, they can send commands to reconfigure the devices in case of an abnormal behavior.

*1) System Setup:* Figure 9 shows an excerpt of the system architecture of the RDS combined with the hardware topology. The 3-tier system is comprised of a perimeter network server (DMZ) running the `RDSConnectionPoint` for the remote access of the industrial devices, an application server with the business components, and a database server running the `Database` component.

The business logic is comprised of a `Parser` component, that processes the input when a status report is uploaded, and `DeviceData` that processes and forwards data. Further, it includes the `DataMiningAndPrediction` component that performs data analysis, and the `DataAccess` component that handles the communication with the database component. The component `ServiceEngineerWebsite` realizes the user interface for the operators to access the status reports.

As in case study I, we determine the performance (using a constant usage profile) and the costs of the system as quantified QAs. We reuse the performance and cost models of a previous work [14].

*2) Experiment execution:* In addition to the performance and costs, we consider security as a potential additional not-quantified QA. In contrast to the previous case study

in this experiment, we assume that security is comprised of two dimensions: The data loss prevention and intrusion prevention. Again, we analyse the trade-offs of higher security vs. performance and costs.

We add a component `Parser'` component that is better in intrusion prevention (*high* vs. *low*) than the previous implementation. Due to the better intrusion prevention we doubled the CPU resource demand for this component as resource demand overhead. Further, we add a new `Database'` component with better data loss prevention capabilities (*high* vs. *low*). Again, we set a higher CPU resource demand for this component. Here, we assume fourfold demands for better data loss prevention. Both components are alternatives to the previous components and are used by PerOpteryx to generate architecture alternatives.

*3) Results:* Figure 10 shows the QAs of all evaluated candidates. Again, we performed 240 iterations evaluating in total 814 architecture candidates. The optimization identified 11 architectures as Pareto-optimal architecture candidates. Analyzing the architecture candidates clearly shows how the results of the optimization can be used for the decision-making process. In the figure, we marked several candidates that show interesting trade-offs: Candidates I and II causes identical costs and level of data loss prevention, but differ in their level of intrusion prevention and response time. If the architect could forgo a higher level of intrusion prevention, she could achieve 1 ms better response time at constant cost. Alternatively, she could decide to accept worse performance, but having better security.

Candidates II and III show interesting results on the trade-off between performance and costs at constant intrusion and data loss level: Improving the response time at only 0.19 ms would imply higher costs by about 16 %.

Often the performance impact of implementing new features remain unclear. Candidates IV and V show the performance impact when implementing a new security feature, such as intrusion prevention: Candidate IV, coming with low intrusion prevention capabilities, has better response time (2 ms less) compared to the high level candidate V.

### D. Discussion

Both case studies demonstrate how our approach can be used to combine quantified QAs together with not-quantified QAs in an automated design space exploration. Case study I shows how such a process helps to find the best trade-off decisions when including not-quantified QAs, such as security and usability. Case study II demonstrates the benefits of our approach from slightly different points of view: It allows to analyze the impact of increasing one quality attribute on others (as in our case study 16 % higher costs at only 0.18 ms lower response times) or even what it means for the response time to implement a new (security) feature.

Even though the values of not-quantified QAs are estimated and therefore can not guarantee the accuracy of using quantified measures, the results of such an optimization ends in the most promising architecture candidates with low overhead. On these few candidates, more time-consuming detailed measurements can be carried out in turn to refine the results. For the design

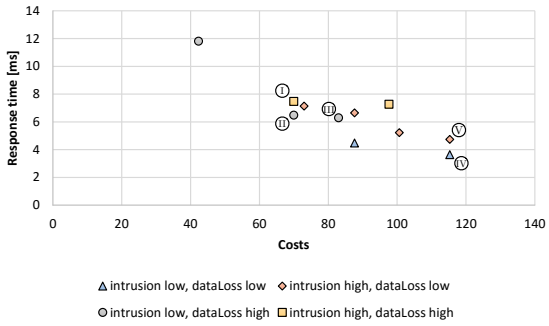


Fig. 10: RDS system: Resulting Pareto-optimal candidates for performance, costs, and security quality

decisions of our example in Section II, the optimization may show that if we add an external service that is performing well and that is also highly usable, the costs of the system may exceed the budget of the project. Based on this information, the project owners could rethink about and prioritize their requirements to make a trade-off decision. One trade-off may result in accepting less performance or usability. Another solution may be to increase the cost limit for a maximum user satisfaction. Alternatively, if the trade-off may be too hard, an architect can, for example, investigate promising candidates more accurately and replace the initial estimate by more precise measures.

## VII. Related Work

The related work to the approach presented in this paper can be classified into three groups. The first group focuses on the area of quantifying quality attributes: Svahnberg et al. showed in [15] an approach that supports the evaluation of different architecture candidates according to the considered quality requirements using a multi-criteria decision method. Their approach uses the Analytic Hierarchy Process (AHP) for a pair-wise evaluation of architecture candidates. Using a set of architecture candidates and quality requirements, their six steps decision process can be used to identify the best architecture candidate according to the requirements. QUPER, proposed by Berntsson-Svensson et al. in [16] is a qualitative approach to estimate quality requirements, such as costs. However, QUPER and the AHP based approach are fully manual processes and therefore very time-consuming when evaluating many architecture candidates. Glinz showed in [17] a value-oriented approach instead of quantifying requirements. He determines the risk that a software system does not satisfy the stakeholders and how to mitigating this risk as cheaply as possible. The approach in [18] demonstrates how to quantify security in component-based systems. It considers several intrusion relevant factors, such as the attacker, his goals, the security of the components, and their mutual interference. A semi-Markov process is used to model these entities. The approach results in a metric, the mean time to security failure, for estimating the degree of security when the attack targets on accessing a certain system component. However, all these approaches either are comparatively time-consuming processes that require many input data, are manual processes or the quality first needs to be broken down to costs. None of the approaches allow to estimate arbitrary and not quantified QAs

and being included in an automated decision support approach together with quantified QAs.

The second group concentrates on the field of architecture trade-off decisions: Kazman et al. showed in [19] a qualitative method for architecture tradeoff analysis (ATAM). It considers multiple QAs and identifies tradeoffs between them. The goal ATAM is to navigate the design space manually, whereas our approach provides an automated search for optimal solutions. Aleti et al. show in [3] ArcheOpterix allowing architecture optimizations for embedded systems considering QAs using an evolutionary algorithm. They use an architecture description language to specify the system architecture to be optimized. ArcheOpteryx results in a set of Pareto-optimal candidates (Pareto-front). An architect can select the best suiting candidate by analyzing the resulting Pareto-front. However, ArcheOperyx requires already quantified QAs for the optimization run. Amyou et al. show in [20] how to use goal models to carry out trade-off decisions on software architectures. For their analysis, they use the goal-oriented requirements language. Their approach supports quantitative and qualitative analysis. But however, for their quantitative analysis they do not consider metrics (e.g., response time), but rely on integer value ranges.

The third group considers requirements prioritization that helps to decide which quality attribute is important: Karlsson et al. show in [21] a comparison of six methods on how to prioritize software requirements. They compared methods from different areas, such as analytic hierarchy process (AHP) methods, but also sorting, and search algorithms, such as bubblesort and binary search tree. The authors characterized these methods to give a user an idea of when using what method. In an industry study they found AHP as one of the most promising techniques. In [22], Karlsson uses, besides other techniques, numeral assignment as one method to assign numbers or symbols to requirements. These symbols are used as a basis to compare the requirements and prioritize them. Herrmann and Daneva compare in [23] methods prioritizing requirements based on benefits and costs. They provide a classification scheme to compare existing methods allowing an architect to select the best suiting approach. However, again, none of the described approaches can be used in automated optimization techniques.

## VIII. Conclusions

We presented an approach to optimize SAs by considering quantified QAs, such as performance, together with not-quantified QAs. Related approaches either allow a quantified or a qualitative consideration of QAs. But often, only a small set of QAs can be quantified, due to time and cost constraints, or lack of existing practical quantification methodologies. In practice, this often means that although detailed models are created for the performance of the system, they can not be set into relation with other, equally important QAs, just because these could not be quantified. In contrast, our approach allows to consider quantified and not-quantified QAs together. This even works for QAs that are either too expensive to quantify, or even lack a methodology to quantify them. For applying our approach we only need an estimate of the component's QAs.

We applied two case studies showing the applicability of our approach and demonstrating its benefits on real-world software systems. Further, we demonstrated how our approach can be used to systematically make trade-off decicions for different quality requirements, even if several attributes are not-quantified. In the course of this case studies, we demonstrated promising results how quantified measures can be combined with not-quantified estimations and how they can be used to select the best architecture candidates. This results can be used to further investigate in promising architecture candidates allowing to focus available resources and filter unpromising candidates, what in turn leads to higher software quality and less development costs.

As further extension of our approach, we will integrate mechanisms to define system-specific evaluation functions that allow the architect to annotate estimations for the same quality dimension to several components and additionally define how these estimations can be aggregated to a single quality evaluation of the system.

## References


[1] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *J. of Sys. & Soft.*, 2009.

[2] A. Martens, H. Koziolek, S. Becker, and R. H. Reussner, "Automatically improve software models for performance, reliability and cost using genetic algorithms," ser. WOSP/SIPEW ICPE'10, 2010.

[3] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya, "Archeopterix: An extendable tool for architecture optimization of aadl models," in *MOMPES '09*, 2009.

[4] F. Bachmann, L. Bass, M. Klein, and C. Shelton, "Designing software architectures to achieve quality attribute req." *SW Proceedings*, 2005.

[5] S. Frølund and J. Koistinen, "QML: A Language for Quality of Service Specification," Hewlett-Packard Laboratories, Tech. Rep., 1998.

[6] R. Reussner *et al.*, *Modeling and Simulating Software Architectures - The Palladio Approach*. MIT Press, 2016, to appear.

[7] A. Koziolek, *Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes*. KIT SP, Karlsruhe, 2013.

[8] G. Franks, T. Omari, C. M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Trans. on Software Engineering*, 2009.

[9] A. Koziolek, "Architecture-driven quality requirements prioritization," in *TwinPeaks*. IEEE CS, 2012.

[10] A. Koziolek, H. Koziolek, and R. Reussner, "PerOpteryx: automated application of tactics in multi-objective software architecture optimization." QoSA-ISARCS 2011, 2011.

[11] Q. Noorshams, A. Martens, and R. Reussner, "Using quality of service bounds for effective multi-objective software architecture optimization," in *QUASOSS '10*, 2010.

[12] X. Wu and M. Woodside, "Performance modeling from software components," in *ACM SIGSOFT Software Engineering Notes*, 2004.

[13] G. Franks, P. Maly, M. Woodside, D. C. Petriu, and A. Hubbard, "Layered queueing network solver and simulator user manual," *Dept. of Systems and Computer Engineering, Carleton University (December 2005)*, 2005.

[14] T. de Gooijer, A. Jansen, H. Koziolek, and A. Koziolek, "An ind. case study of performance and cost design space explor." ser. ICPE '12, 2012.

[15] M. Svahnberg and C. Wohlin, "An investigation of a method for identifying a software architecture candidate with respect to quality attributes," *Empirical Software Engineering*, 2005.

[16] R. B. Svensson, Y. Sprockel, B. Regnell, and S. Brinkkemper, "Setting quality targets for coming releases with quper: an industrial case study," *Requirements Engineering*, 2012.

[17] M. Glinz, "A risk-based, value-oriented approach to quality requirements," *IEEE Software*.

[18] A. Busch, M. Strittmatter, and A. Koziolek, "Assessing Security to Compare Architecture Alternatives of Component-Based Systems," in *QRS'15*, 2015.

[19] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," in *IEEE Intl. Conf. on Eng. of Complex Computer Systems, 1998. ICECCS'98.*, 1998.

[20] D. Amyot, S. Ghanavati, J. Horkoff, G. Mussbacher, L. Peyton, and E. Yu, "Evaluating goal models within the goal-oriented requirement language," *Int. J. of Intelligent Syst.*, 2010.

[21] J. Karlsson, C. Wohlin, and B. Regnell, "An evaluation of methods for prioritizing software requirements," *Inform. and SW Technology*, 1998.

[22] J. Karlsson, "Software requirements prioritizing," in *Proceedings of the Second International Conference on Requirements Engineering*, 1996.

[23] A. Herrmann and M. Daneva, "Requirements prioritization based on benefit and cost prediction: an agenda for future research," in *IEEE Intl. Requirements Eng. (RE)*, 2008.