Integrating Quality Modeling in Software Product Lines

Joerg Bartholdt
Corporate Technology
Siemens AG
Munich, Germany
joerg.bartholdt@siemens.com

Roy Oberhauser
Computer Science Dept.
Aalen University
Aalen, Germany
roy.oberhauser@htw-aalen.de

Andreas Rytina
itemis
Munich, Germany
andreas.rytina@itemis.de

Marcel Medak
FNT GmbH
Ellwangen, Germany
marcel.medak@fnt.de

Abstract— **Due to the large number of possible variants in typical Software Product Lines (SPLs), the modeling of, explicit knowledge of, and predictability of the quality tradeoffs inherent in certain feature selections are critical to the future viability of SPLs. This article presents IQSPLE (Integrated Quality Software Product Line Engineering), an integrated tool-supported modeling approach that evaluates both qualitative and quantitative quality attributes without imposing hierarchical structural constraints. This contributes to better traceability; annotation; constraint enforcement; and quality attribute trade-off analysis - depicting overall product quality impacts on-the-fly. The approach is used in an eHealth SPL scenario, with the results showing that this approach is promising for effectively integrating quality attributes into SPL engineering in conjunction with (UML-based) artifacts.**

Keywords - variability; software product lines; quality modeling; feature modeling

I. INTRODUCTION

SPLE seeks to foster a systematic reuse of software assets for different but similar software products (typically within a domain). The general approach captures the commonalities and variability of the products in the product line and splits the development into domain (commonalities) and application (additional individual features for the final product). Products are created by integrating common artifacts (usually a platform) and optionally configuring them with product-specific artifacts [3][4].

Significant feature-oriented work and methodologies such as Feature-Oriented Domain Analysis (FODA) [5], FeatuRSEB [6], PuLSE [7] are well known for domain analysis and variability modeling for SPLs. However, for a potentially large set of possible variants, a significant aspect yet to be sufficiently addressed is the consequences of choices on the end qualities exhibited by a variant. An SPL engineer is faced with many more quality-related unknowns than a software engineer for a common single application software architecture. While various approaches for combining quality modeling with SPL engineering (SPLE) exist, previous work does not provide an integrated tool-supported approach with both qualitative and quantitative quality attributes (Q-attributes) that are explicitly considered in the variant derivation process without imposing structural constraints such as a hierarchical structure. In this problem space, the tool-supported IQSPLE method contributes trade-off analysis, traceability, annotation, and constraints enforcement of quality attributes during selection. Our previous work in [1] was extended to directly integrate solution space quality modeling with Unified Modeling Language (UML)-based artifact annotation support for variability and quality annotations as well as aggregated quality evaluation capabilities.

Considering the need for trade-off analysis, the distribution of quality attributes can vary significantly in software products, as shown in [8] that studied 24 ATAM (Architecture Tradeoff Analysis Method) evaluations. Such quality attributes are often not fully and systematically captured in prose. Even if formal models like the OMG (Open Management Group) UML-related QoS profile are used, an automatic aggregation ability is requisite to benefit most from a formal description. IQSPLE contributes methods and tools to immediately derive the quality attribute values of a given product instantiation.

Because qualities in SPLs often describe crosscutting concerns, the definition of qualities in the problem space is generally not linked to the solution space, resulting in a lack of traceability. IQSPLE contributes traceability via a formal linking that is used to calculate the quality attributes from the selection of product variations via the properties of assets in the solution space. This also supports the detection of quality issues for certain SPL variants that can be used in narrowing tuning efforts to the relevant solution artifacts.

Typical feature-oriented tooling concentrates on functional features; quality constraints are, if at all, modeled as simple XOR on features and thus remain purely in the problem space. IQSPLE enhances current feature modeling with support for the annotation of solution components with quality properties and arbitrary aggregation functions. By linking to the features, automatic constraint checks on given quality requirements can be executed. To enforce a common understanding and enable automatic calculation in the problem and solution space, a formal quality-capturing model for crosscutting as well as localized quality attributes is necessary.

This paper is structured as follows: Section II describes an e-Health scenario with the ensuing requirements that initiated the research. Section III describes the IQSPLE solution approach, which is then illustrated via an eHealth SPL scenario in Section IV. The solution is evaluated in Section V, with Section VI discussing related work. A conclusion and references follow.

II. E-HEALTH SCENARIO AND REQUIREMENTS

For illustration purposes, a simplified eHealth problem scenario that motivated this research is used. Patients are referred to other clinics due to their specialization (surgery, physical therapy, imaging, etc.). In the past, computed tomography images, clinical findings, etc., were given to the patient in the form of a printout or CD to take to the next treatment. This was error-prone, not all information was necessarily available at the next treatment location, and one was not sure that the data was current.

The eHealth scenario describes a clinic chain that wants to introduce SEPDE (software system for electronic patient data exchange) between organizations. The existing hospital information systems (HIS) are supplemented with SEPDE. The chain consists of ten hospitals where eight have the same HIS product and two have individual solutions.

Figure 1 shows a reduced feature tree of the SEPDE SPL. Integration with existent HIS, which is a key feature of the product line, can be achieved with three different techniques: web services, CORBA or message-based. The message-based approach allows for two different options: A high-throughput, but expensive commercial one or a slower, but license-free open-source variant.

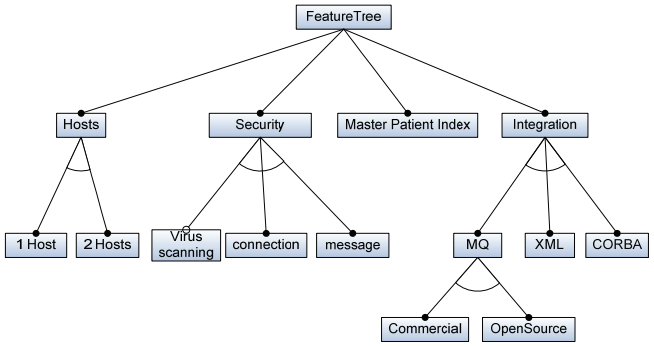

Figure 1. Conventional feature tree.

The development effort for creating the adapter to connect to existing HIS differs: XML-based web services are perhaps easier to develop, but carry greater performance overhead compared to binary protocols. The CORBA-based binary integration makes the final solution better in terms of latency and throughput, but development efforts are higher due to its complexity. A message-based approach increases integration flexibility and scalability, yet development complexity increases due to asynchronicity and lack of object-oriented remote method calls while longer latencies can be expected due to the centralized message broker.

For security and privacy, confidential connection (mapped to SSL connections), message-based encryption and signature (allows secure audit records), and virus scanning is supported. While all decrease the overall performance, the former two are necessary if no VPN exists between the sites; the latter additionally results in ongoing costs for continuous updates.

For higher availability of the system, a two-host-solution can be instantiated. The session state must then be replicated between the hosts so it exists should one of them fail.

This simplified scenario illustrates the relevance of quality attributes (e.g., performance, price, security) to the choice of specific features. The customer may not have exact requirements (e.g., 'use case 15' must be performed in less than 1.5 sec). However, the customer may be able to trade quality attributes against each other or functional features (e.g., 1.5 sec is achievable with the commercial MQ with a 5000€ license, whereas with the open-source solution it is 1.8 sec – which might be acceptable).

Each functional feature influences to some degree all system quality attributes, making the manual tracking of quality attributes difficult. Common feature trees contain functional features that are selectable individually (with a few constraints between each other), while system quality attributes are a crosscutting concern that changes with each (de)selection of a feature. Moreover, the quality correlations are often not expressible in simple constructs. Feature model constraints could (de)select features automatically, causing an entire set of quality attribute values to change at once.

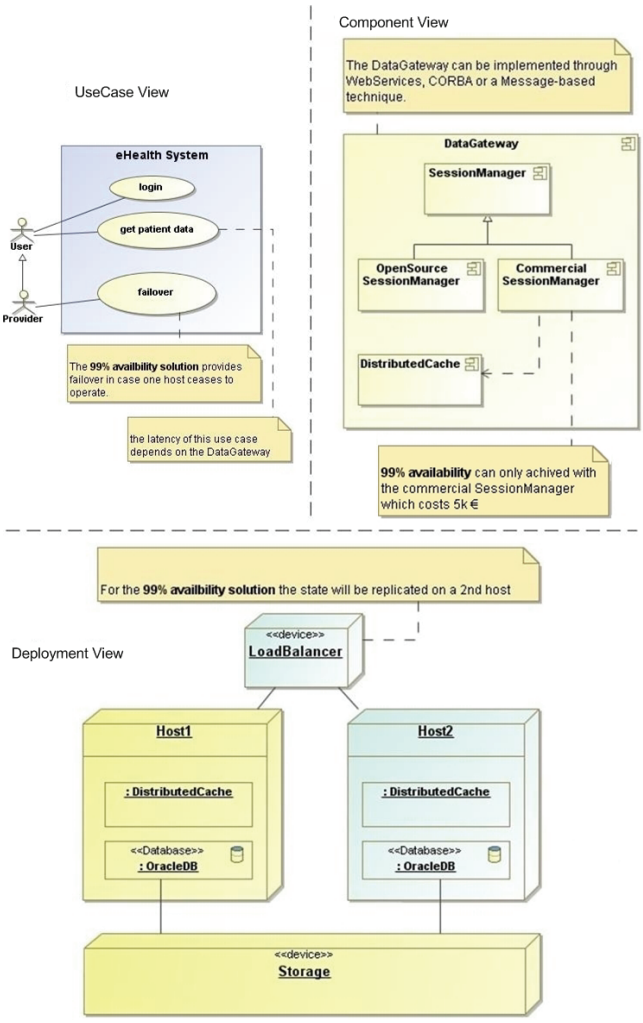

Figure 2. Example models affected by variability.

Qualities of the solution manifest themselves in different views. Many quality constraints can be described on the

component level, e.g., the usage of a specific session manager implementation correlates to the availability of the system, see Figure 2. Others become apparent in the deployment view, e.g., the availability of the solution depends on the number of redundant elements or even the existence of components like a load-balancer, but only make sense when more than one host exists. Certain use cases are valid only or change their nature depending on the features selected or on quality parameters. The resulting use case view can be used to generate development specification documents or user manuals. An additional benefit of a consistent and integrated usage of modeling of the views is the automatic traceability of features in models and specifications.

Considering non-trivial SPLs, the impact on quality attributes is not foreseeable for the SPL engineer, thus there is the need for methodology and tool support. As quality is usually not exactly defined by the customer beforehand and requirements may change, quality support is needed during the selection process.

A. Requirements

The following requirements on the methodology (M-requirements) and tool support (S-requirements) are deduced from the scenario:

M1: *qualitative values*. It must be possible to define quality values such as low, medium, or high in cases where a quantitative expression is not feasible or would be too expensive to measure. The quality attribute must be definable in the solution space as a property of an artifact and in the problem space as a non-functional requirement of the customer. Because non-functional customer requirements depend on design details, it must also be possible to link the qualities between the problem space and the solution space.

M2: *quantitative values*. It must be possible to define quantitative values (e.g., memory footprint, response times) in order to calculate the resulting quality values of the instantiation. Here also, the quality attribute must be definable in the solution as well as the problem space.

M3: *algorithms for calculating the resulting attribute values*. The methodology must support the definition of a calculation algorithm ("aggregation function") for the resulting quality value. This applies to quantitative as well as qualitative values (however, it may be less intuitive for qualitative values).

M4: *presentation as feature*. To support configuration and selection of qualities for a product instance, qualities can be presented in a separate quality tree like features, or alternatively within the feature tree, e.g., when qualities are relatively straightforward and the maintenance of a separate quality tree would seem contrived.

M5: *artifact annotation.* Quality attributes are treated as a first-class modeling citizens, and modeling and other artifacts can be annotated wherever appropriate. The annotation mechanisms shall follow a standardized mechanism to foster reuse and community acceptance.

S1: *calculate the quality values of a given variant*. The quality attributes that result from the selection have to be calculated (ideally on-the-fly), to give immediate feedback and let the users realize what changes in quality values a change in features results in.

S2: *determine the set of possible variants*. Given quality constraints during the selection process, the tooling shall determine the valid variants.

S3: *constrain the selectable features*. Quality attribute requirements shall be definable in advance and during feature selection, with those features not selectable whose selection would impair the required quality.

S4: *visualization of quality values*. From a customer perspective, multiple quality attributes may be of interest and may differ between customers. Thus, the tooling shall support an appropriate yet configurable visualization of the resulting quality values.

S5: *quality modeling integration in solution space*. Consistent, gap-less usage of modeling techniques, especially for quality modeling, leverages available tooling. All views on the solution space of the product line will more or less be influenced by quality attributes. The tooling must handle all of these in a consistent way, thus consistent meta-models must be applied on which transformers, generators, evaluators and viewers rely.

S6: *reuse of artifacts.* Automatic evaluation of quality attributes requires a formal description of quality properties and correlations. These formal descriptions can also be used for other purposes, e.g., for generating product specification documents, manuals, etc.

S7: *traceability support.* Generated artifacts should carry dependent tracing information, e.g., the "administration of multiple hosts" chapter in the manual depends on the selection of a high availability solution.

III. SOLUTION

To address the aforementioned requirements, IQSPLE integrates quality attributes in the solution artifacts, maps the feature and quality selection in the problem domain to the associated solution artifacts, and collects and evaluates the quality attributes. With appropriate aggregation functions, the quality attributes of the product instance can be automatically evaluated and displayed in the selection configuration. The various elements of the IQSPLE methodology are described below.

A. The IQSPLE Process

The process is depicted in Figure 3. SPL domain engineering involves:

1) *Requirements Analysis.* Through the analysis of the problem domain, common and variable feature and quality requirements are collected.
2) *Feature variability and quality variability modeling.* In addition to the typical feature modeling in a feature tree (e.g., using the Compositional Variability Management (CVM) framework, a separate quality tree is used to model the quality attributes and their value types, e.g., memory footprint in MB, latency in 'use case 15' in ms). It is assumed that components can be assigned quality

attribute values based on a specification or measurement. Note that discrete values need not be ordered.

The resulting quality tree can serve as a basis for selecting non-functional requirements. As appropriate, elements in the trees can be linked to other trees (e.g., a selection in the quality tree might deselect certain features in the feature tree) and to the UML models.

For automated synchronization support, UML vendor-specific APIs can be used to allow changes to the (quality-annotated) models to be automatically reflected in the trees (e.g., certain quality options might disappear if no longer supported in the solution models). This supports M1, M2, and M4.

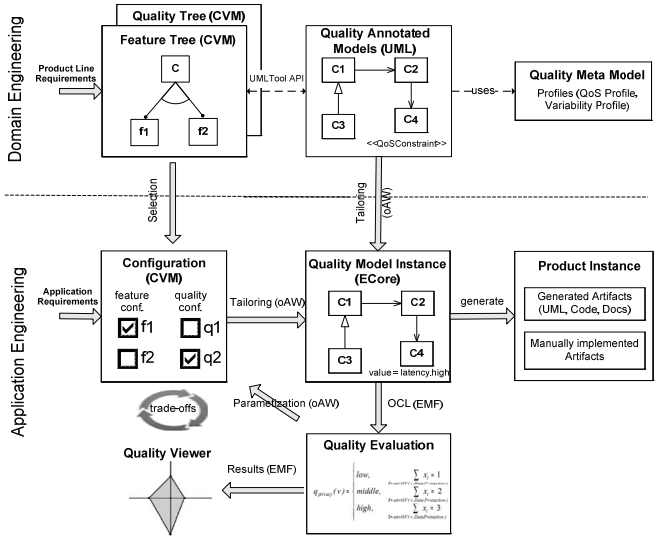

Figure 3. IQSPLE process.

3) *Modeling of solution artifacts including quality-annotations.* A quality meta-model such as the UML QoS Profile, variability profile, etc., is used to allow solution space models (i.e., software artifacts) to be annotated with quality attribute names and values. Note that each element can have multiple quality attributes assigned and these elements can be linked with features (e.g., memory consumption, use-case-specific latency, scalability properties), but not all components must be assigned all attributes. E.g., all components will affect memory, but not all will influence the latency in use case 15. This supports M1, M2, M5, and S5, S7.

SPL application engineering involves:

4) *Configuration.* A product configuration is selected based on feature and quality requirements (e.g., using CVM). This supports S2 and S3.
5) *Quality Model Instance generation.* Based on the configuration information and using UML tooling (e.g., openArchitectureWare (oAW)), a tailored Quality Model Instance is generated (e.g., in ECore).
6) *Quality evaluation.* OCL statements in the Quality Model Instance are used to evaluate the resulting qualities. A quality (aggregation) function is defined to support M3 and S1. To calculate the overall quality of the resulting product instance, the quality attributes of all selected components must be aggregated. In simple cases, this can be a *sum*-operation (e.g., memory footprint, latency) or *min*-operation (e.g., security behavior is as good as the weakest component). Complex operations are also possible, e.g., encryption depends on message length which depends on the selected features, so influence may be expressed in factors such as "encryption increases latency of use case 15 by 50%". For quality-based attributes, the aggregation function may count the number of occurrences, the most frequent wins, or calculates an average over ordered elements.
7) *Quality validation.* Based on the quality viewer, the qualities achieved are validated by the user or, based on tradeoffs, the configuration is adjusted as necessary. This supports S4.
8) *Product Instance Generation.* Once the configuration has been validated, artifacts are generated (e.g., UML models, code, documentation including specifications, and user manuals), and manual artifacts are implemented. This supports S6.

B. *Variability*

Negative variability. Negative variability starts from a maximal description (e.g., a UML model containing all possible elements of the product line) and deletes the elements that are not connected to selected features [2]. By this reduction, the final model of the selected product instance will be the result.

Depending on the selected features, model elements can be removed to derive different product instances. This is reflected in the model by tagging the different types with the stereotype `<<Variation>>`. The condition for which it is generated for the product instance is defined by the tagged value `{feature = "any feature condition"}`. This indicates to the generation process that the elements associated with the feature condition are generated if the condition evaluates to true, otherwise they are removed.

This is called negative variability since the starting point is a superset of the model definition and the unnecessary elements are stripped away according to the features selected. [2] discussed negative variability in class diagrams to model data structures of product lines and generate the data model for a selected product instance. In order to integrate this approach with quality modeling, this mechanism was extended for other diagram types. E.g., the results of the quality "scalability" will be seen on a deployment diagram that contains one, two, or more hosts. Another example is the deletion of a use case in a use case diagram because a certain feature was deselected. Use case diagrams can form the basis for product specification and manuals, which should also adapt automatically to selected features for the product instance.

Structural variability. Structural variability describes a change in the model dependent on some feature selection [2]. The element is already contained in the model, but its structure (type, cardinality, association) may vary. Structurally changing a UML model is achieved by adding

the stereotype <<modify>> to the elements that should be structurally changed and by setting predefined tagged values. Possible tagged values are, e.g., feature, type, cardinality, name, and initialValue.

In the resulting model, the corresponding property is changed. This can also be used to redirect associations by changing the type of the association.

C. Integrity Constraints

Constraint checking and their languages such as the OCL (Object Constraint Language) in UML are a known and powerful capability for assuring modeling correctness. Since SPLs typically support a large number of variations and quality aspects are typically crosscutting concerns that affect multiple models, constraint capabilities should be applied at the most appropriate points across the tooling.

For instance, feature/quality modeling constraints can be utilized to determine the validity of a certain combination of features or qualities (e.g., CVM provides a proprietary language to specify feature constraints). Instance tailoring constraints can be applied to check conditions (e.g., ensuring that the domain and feature/quality models are consistent) before or during the tailoring process as well as the artifact generation process.

D. Quality Functions and Annotations

In order to enable the evaluation of qualities of a product instance, mathematical functions are used to aggregate quality attributes. These functions are defined as relations between a valid variant *v* and a value *x*, where *x* represents the state of a specific quality.

$$q_i : v \mapsto x \tag{1}$$

To access attribute values of different artifacts, two additional functions are defined. The function *attribE* returns a single attribute value of a specific solution element (e.g., concrete component) and requires a valid variant *v*, a specific element *e* and the intended attribute *a*.

$$attribE : v \times e \times a \mapsto x \tag{2}$$

The second function *attribV* returns a vector of all attribute values of a variant that match the provided attribute name. This provides access to values and can be used if no exceptional conditions must be taken into account.

$$attribV : v \times a \mapsto \vec{x} \tag{3}$$

Note that there are no limitations on using different value ranges for aggregations, as long as a reasonable aggregation function for the quality can be determined. Numbers for calculating memory footprint are just as possible as using low, middle, high values to assess, e.g., security aspects.

As an example, privacy could be defined as follows:

$$q_{privacy}(v) = \begin{cases} low, & \sum_{\vec{x}=attribV(v,DataProtection)} x_i = 1 \\ middle, & \sum_{\vec{x}=attribV(v,DataProtection)} x_i = 2 \\ high, & \sum_{\vec{x}=attribV(v,DataProtection)} x_i = 3 \end{cases} \tag{4}$$

By modeling restrictions (Figure 1) on feature associations as constraints, M4 is supported. A constraint c_i is defined as a relation between a variant *v* and one element of the set *{true, false}*.

$$c_i : v \mapsto \{true, false\} \tag{5}$$

This allows a definition of, e.g., performance requirements based on predefined quality functions.

$$c_{performance}(v) = \begin{cases} true, & q_{latency}(v) \le 300ms \\ false, & else \end{cases} \tag{6}$$

The constraints are used as a way to filter out remaining variants that violate given requirements. For deselection functionality support for S2 and S3, IQSPLE inspects each feature beneath a selection line in the feature tree and decides if a feature selection violates the given requirements. In order to decide feature selectability, IQSPLE distinguishes three fundamental cases. A feature is:

a) *not selectable* if it does not occur in any remaining variant,
b) *selectable* if it occurs in every variant,
c) or *in combination selectable* if it occurs in some variants but not in all.

Cases *a* and *b* are trivial. However not every consequence of a selection can be predicted, especially if there are still open selections that affect the fulfillment of constraints. Thus, IQSPLE uses Case *c* to indicate that a feature selection possibly can only be made dependent on further feature selections.

An example is shown in Figure 4. To illustrate the process, a constraint is defined that forces a selection of at least five features. Initially all variants are derived that fulfill the constraint. Subfigure (1) shows the root feature tagged in green, which means that at least one variant of the feature tree matches the constraint and that feature f_0 is contained in the set of the valid variants. Green features must always be selected.

In (2) f0 is selected and the selection cut is moved below f0. Here f1 is tagged in green and f2 in red. It is obvious that f2 will always be tagged in red because of the alternative relation to f1, which is an invalid selection since with f2 maximally four features can be selected. Consequently, f1 must be selected as shown in (3). In this case f3 is selectable and f4 in combination selectable. In (4) f3 has been selected and f4, f7 and f8 are in combination selectable. While the

current selection does not fulfill the constraint of at least five selected features, it is recognizable that a combination of the three remaining features would. If the current selection is a valid variant and fulfills the constraint, the selection of a blue feature is optional. For the case that a current selection does not fulfill a constraint, it is necessary to select further blue features.

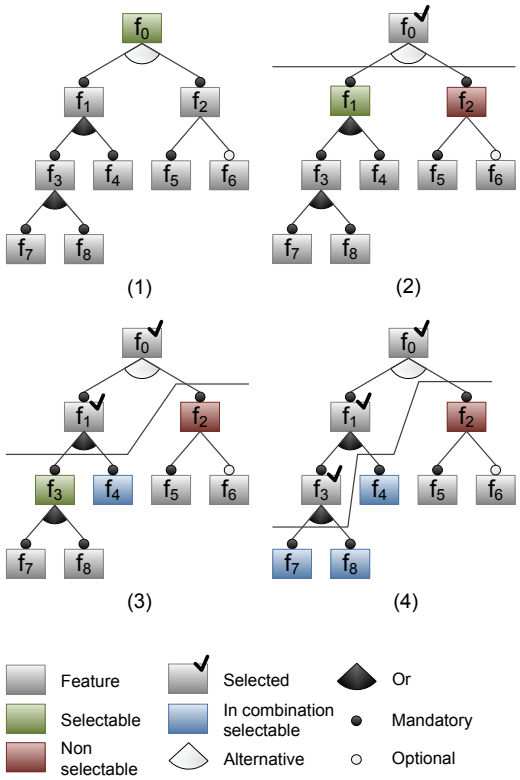

Figure 4. Selection process example.

Thus the IQSPLE process supports the handling of fixed requirements and, depending on the stakeholder's perspective, one can see either which subset of features still fulfill the requirements or if the selection of a certain feature does not.

In the current implementation, all quality functions were defined in OCL due to its expressiveness, UML integration, and standardization, as shown in Figure 5. In the diagram, four Quality of Service (QoS) characteristics are shown for latency, security, costs, and availability. To annotate certain constraints as quality aggregation functions, the OMG "UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms" [9] was extended with the `<<aggregateFunction>>` stereotype. Each `QoSCharacteristic` can have at most one `aggregateFunction`. To allow the retrieval of feature expressions from within a `QoSCharacteristic`, `QoSCharacteristic` inherits from the class `AbstractQoS`. Static methods are defined in the `QoSCharacteristic`, which can be additionally used for the definition of constraints.

The range of attribute values is determined by the `QoSCharacteristic`. For instance, costs are usually positive numbers that are summed, while usability could be either decreased by adding more components to administrate, or increased by a module that presents role-based administration user interfaces.

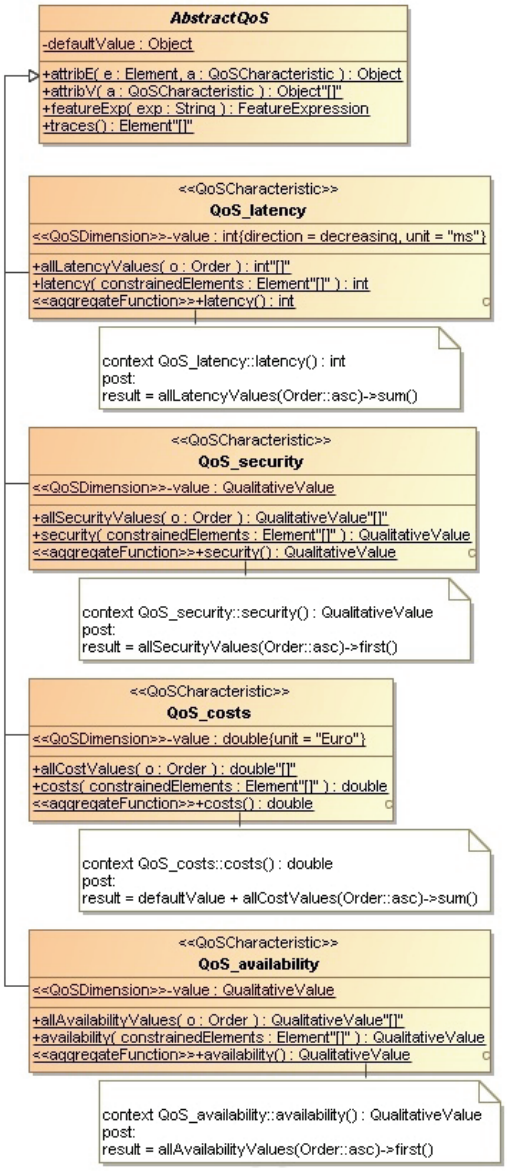

Figure 5. QoS characteristics with OCL quality aggregation functions.

In the OCL code of Listing 1, the aggregation function for latency is shown.

Listing 1

```
context QoS_availability::availability() :
QualitativeValue
post:result = allAvailabilityValues(Order::asc)
              ->first()
-- same query with the generic method attribV:
-- result = attribV(QoS_availability)
--  .oclAsType(QualitativeValue)
```

```
--  ->sortedBy(integerValue())->first()
```

The method `allAvailabilityValues(..)` returns an array of the results of all the availability constraints. The input parameter defines the sorting order of the array. The parameter `Order::asc` is for ascending order. The method call `first()` returns the smallest availability value. Alternatively, the call could have been realized with the method `attribV`, as described in the comments. The return values would then need to be cast to the specific type and sorted.

E. Quality Evaluation

The Quality Evaluator calculates the quality values, which utilizes the quality model instance created by the tailoring process. Impacts on qualities are evaluated by executing the OCL, thus calculating the quality attribute values of the selected artifacts for the product instance and aggregating them to an overall value. To support S4, after the quality value calculation and aggregation process, the assessed variant aggregated qualities are presented to the user as a spider chart as shown in the Quality Editor of Figure 6. This process is triggered every time a user changes a feature selection. The effect of a feature selection on particular qualities can thus be dynamically observed in the change to the chart during selection. This is helpful especially in trade-off situations.

Via this mechanism, the attribute values for a specific variant can be concretely defined in the solution space. For instance, a timeout configuration setting can be dependent on the combined latencies of the selected message component (commercial or open source).

Additionally, in order to ensure that the configured variants achieve the required qualities, during configuration the selectability of certain features and qualities are dynamically grayed-out (unselectable) based not only as hitherto on some abstractly modeled constraint/dependency between feature trees or tree elements, but on the impact evaluation of a specific feature or quality selection on variants (configuration feedback). For example, if a user defines that there is a budget of 10000€ for licenses, then all (aggregated) features are grayed out that are dependent on components that do not meet this requirement. At the time of de(selection) of a feature, the resulting configuration is verified against the quality requirements. In case the configuration does not meet the requirements, the user can choose alternatives. Currently a brute-force algorithm is used to recursively evaluate the remaining variants in the tree. Dependent on the number of variants, the automatic assessment of possible variants can take significant computation time. Therefore, typically not every variant is evaluated, but once a user-configurable limit of valid variants is found, the automatic assessment is halted and the alternatives are suggested to the user. If no valid variants are found within some user-defined time limit, the user is required to select additional features to further limit the variant space and thereby shorten the computation time.

As shown in Figure 6, the Quality Editor presents the aggregated quality values of a variant and provides information about the location and number of quality constraints that affect a certain quality attribute. For example, in Figure 6, the user has selected the quality attribute latency (get patient data) and, in the description which contains static and dynamic text, the user sees that most latency constraints are located in the component view.

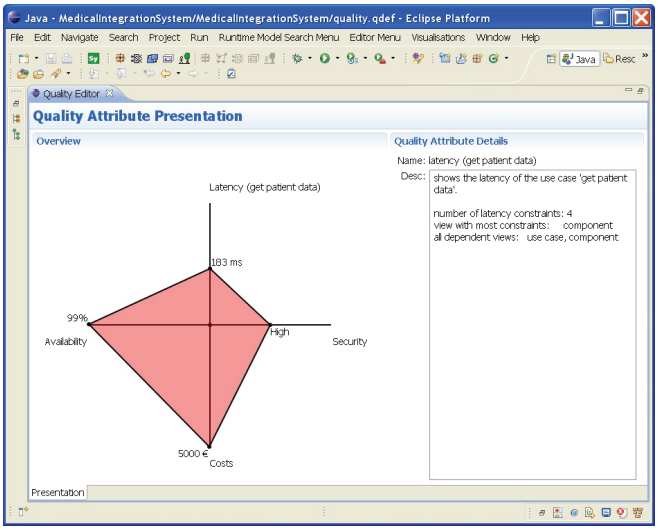

Figure 6. Quality editor.

The attribute values of a model element can also be dynamically calculated or become part of a variant, e.g., as a function of changes in the dependencies and selections.

IV. E-HEALTH SPL EXAMPLE

To illustrate IQSPLE, the E-Health example of Section II will be used applying the process described in section III.A.

1) Requirements Analysis

This is assumed to have been completed in this simplified example.

2) Feature variability and quality variability modeling

The feature tree from Figure 1 is now split into a feature tree (Figure 7) and a quality tree (Figure 8).

The single-host / dual-host features are removed from the feature tree. The customer should not select whether (s)he wants one or two hosts, which was modeled as a feature. Instead, the resulting quality of the solution is defined by the customer, in this case 95% or 99% availability. The dual-host solution is a solution and, as such, not of primary interest to the customer (it might be from an administrative point of view later).

The headline “security” is split. Virus scanning is definitely a customer visible (external) feature and, as such, located in the feature tree. The two other options “connection” and “message-based” were for deciding the security property if the communication can be unprotected, e.g., if the systems are interconnected via VPN. Connection-based security was implemented by using SSL, thus authenticating sender and receiver and encrypting the data in transit, but only a message-based signature results in

auditable messages because the messages together with the signatures can be archived as-is in an audit trail record and the signature can be verified again later. Thus, the customer is primarily interested in the resulting quality properties, e.g., if he has to obey specific regulations for auditable records, but does not care about the actual implementation necessary to achieve this. Security is an example for qualitative values (see requirement M2) with an order ("none" < "confidential" < "auditable").

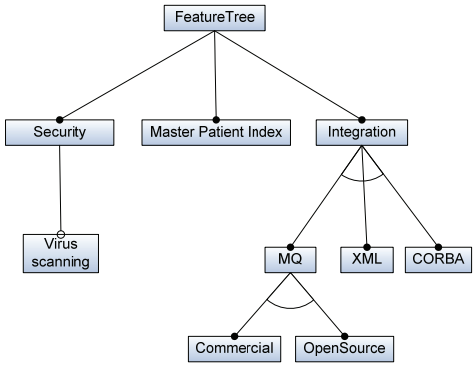

Figure 7. Feature Tree.

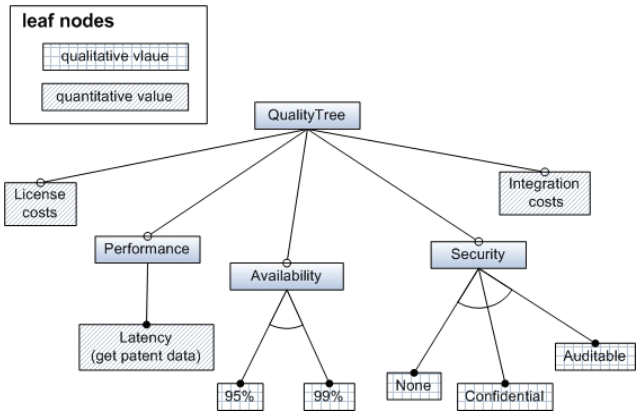

Figure 8. Quality Tree.

Note that a group of features can create additional value, e.g., if the http protocol is used for the user interface and DICOM protocol for image data retrieval, the complete solution is "confidential" only if both protocols provide confidentiality mechanisms, e.g., SSL. The aggregation function would be:

$$q_{security}(v) = \begin{cases} auditable & \text{HTTPcomp.security} == autitable \wedge \\ & \text{DICOMcomp.security} == confidential \\ confidential, & \text{HTTPcomp.security} == confidential \wedge \\ & \text{DICOMcomp.security} == confidential \\ none & else \end{cases}$$

3) Modeling of solution artifact including quality annotation.

The diagrams in Figure 9 through Figure 11 contain a (simplified) super-set of all possible artifacts annotated with constraints on the selected features and qualities. For example, Host2 in the deployment view is marked as `<<Variation>>`. The condition is `{featureExp = "99%"}`, thus the element "Host2" will vanish if the feature is not selected. The same is true for the load balancer. For details on negative variability, see [2].

Additionally, the quality attributes are annotated according to the OMG QoS profile [9] as described in Figure 5.

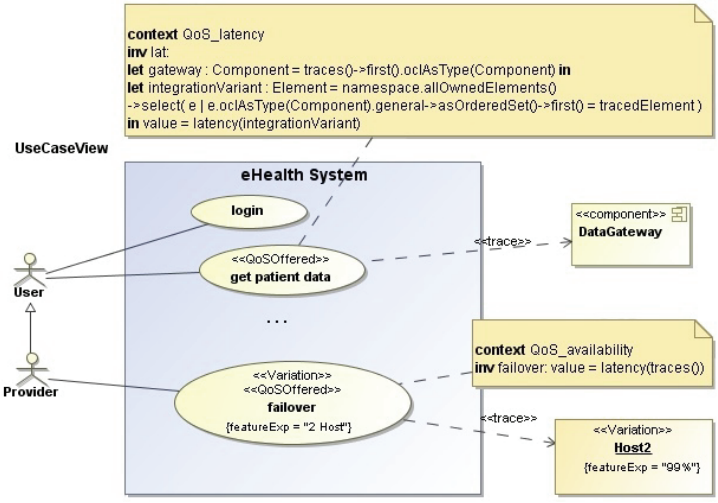

Figure 9. Quality-annotated use case model.

4) Configuration

For a fictitious customer, the product instance as shown in Figure 12 is selected. The customer chooses a message queue as the most flexible technology for integration, which is also the most future-proof variant. To save costs, (s)he selects the open-source implementation. For consolidation of patient records, (s)he wants the new system to have an MPI (Message Passing Interface). He/she selects virus scanning because all systems are connected to the internet, which is how the data exchange is routed.

From a quality perspective, (s)he emphasizes the availability of the system and selects 99%. Since no VPN is in place, (s)he selects confidential data exchange, but (s)he has no requirement for auditable signatures. The customer does not set any other quality constraints in the quality tree as shown in Figure 13.

5) Quality Model instance generation.

This step generates an internal representation of the quality model to be used during the quality evaluation. For performance reasons, this quality model instance only contains the quality constraints, quality aggregate functions, and feature expressions. Based on this simplified model, the Quality Evaluator calculates the quality attribute values for all constrained elements and then aggregates the resulting values to overall quality values.

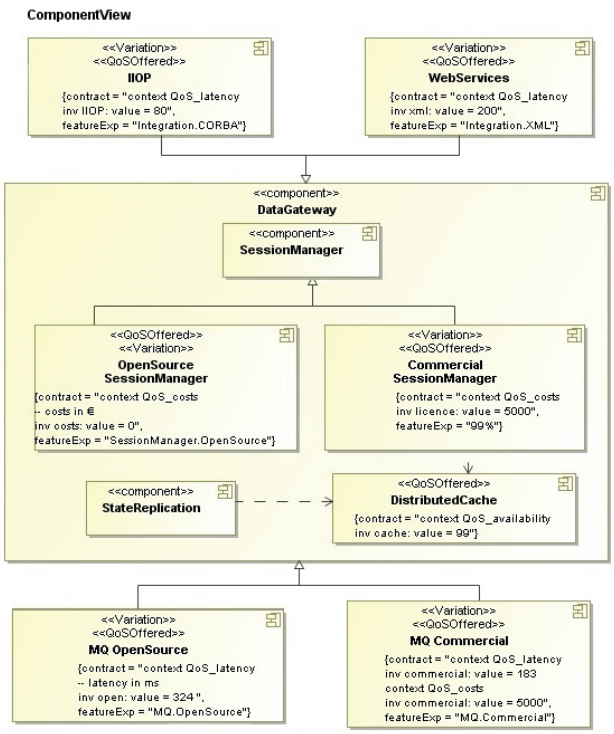

Figure 10. Quality annotated component model.

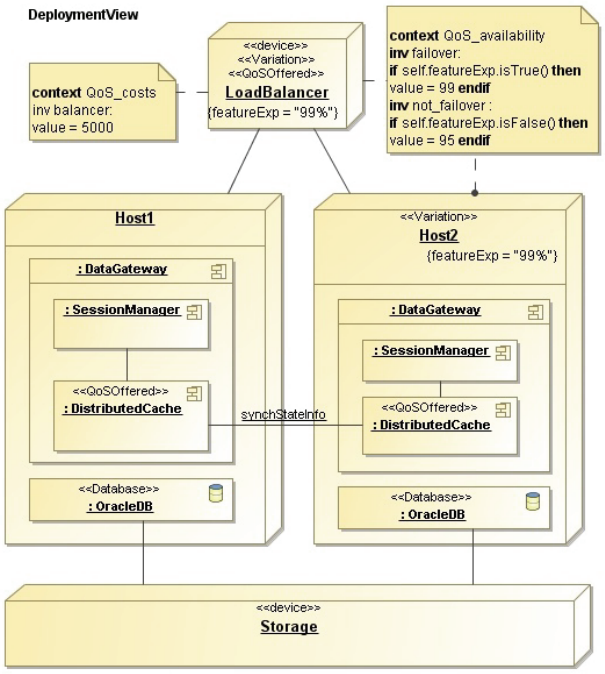

Figure 11. Quality annotated deployment model.

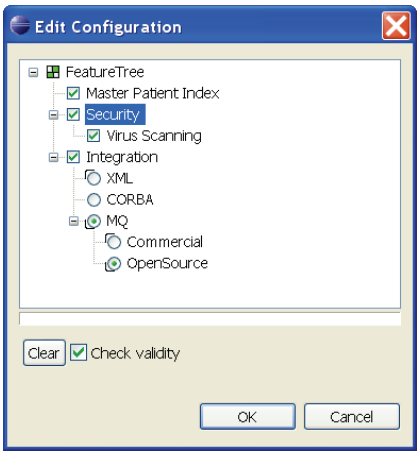

Figure 12. Selected product variant.

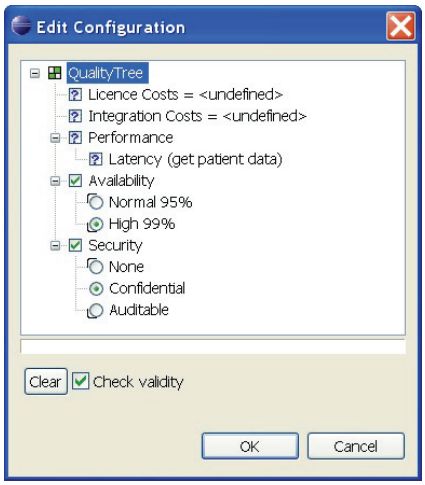

Figure 13. Initial quality tree defined by the customer.

6) Quality evaluation.

Due to the selection in the feature and quality tree, the aggregation functions can aggregate the overall quality attribute values of the product instance (see Figure 17 and 18 for the resulting product instance artifacts). E.g., the license cost (a quality attribute referred to by requirement M1) is determined by the sum of the existing artifacts on all diagrams (see formula in `QoS_costs: allCostValues(Order:asc)->sum()`):

Base	30,000
Load Balancer	5,000
Commercial session manager	5,000
Total	40,000

The evaluation of the quality attributes is shown in Figure 14.

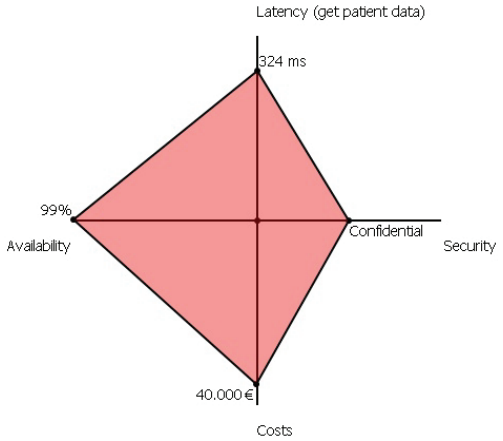

Figure 14. First quality spider chart.

7) Quality validation

The customer realizes that the latency of the use cases, represented by latency for "get patient data," is unacceptable. Modifying the messaging software provider from "open-source" to "commercial" results in the quality spider chart depicted in Figure 15, showing that the price increases to 45000€, but now the latency reaches an acceptable level.

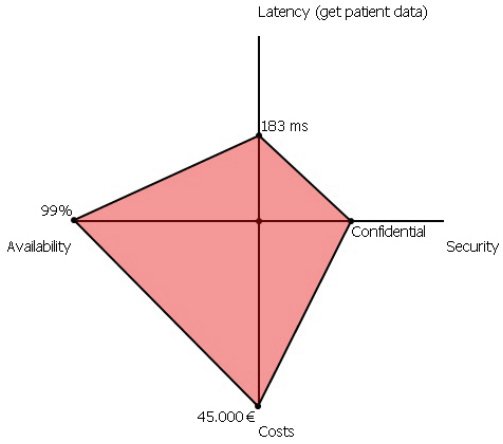

Figure 15. Final quality spider chart.

8) Product Instance Generation

The resulting artifacts are depicted in Figure 16, 17, and 18.

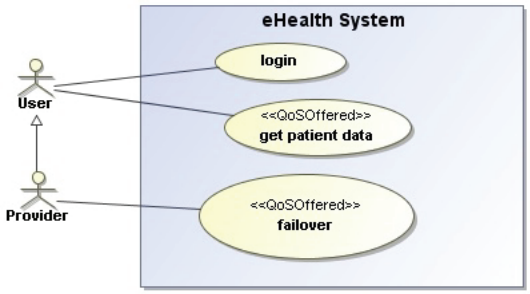

Figure 16. Tailored UseCase View.

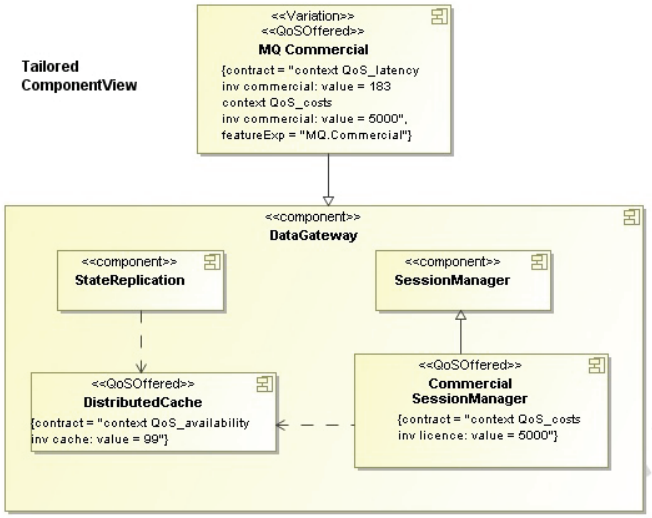

Figure 17. Tailored Component View.

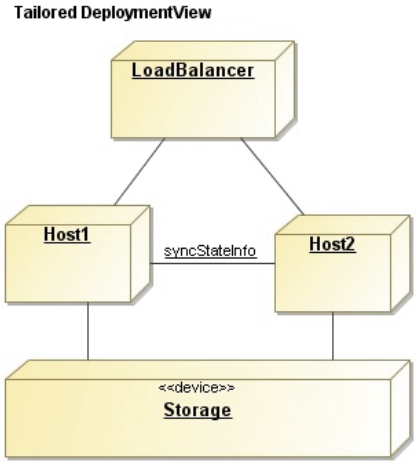

Figure 18. Tailored Deployment View

The use case Failover is included, thus the user manual will contain the section about administration and monitoring of the high availability solution.

The deployment diagram shows dual hosts with a load balancer required to fulfill the 99% availability. From the deployment diagram, the bill of materials can be derived and will now contain dual hosts and a load balancer, which has to be ordered from a 3rd party provider.

The component diagram includes only MQ as an integration provider and a distributed cache that is necessary for the multi-host deployment for sharing the state across multiple hosts.

The eHealth scenario exemplified how IQSPLE facilitates greater thought to and usage of quality in the domain and application engineering stages. First, customers can make decisions about required qualities based on facts instead of subjective estimations from the SPL engineer. Customers are also able to see how a decision affects particular qualities and if the consequences of a decision are acceptable. The SPL engineer benefits since thought to

system qualities are explicitly stipulated, which can help to improve the overall quality of the SPL architecture. In case a quality requirement is not fulfilled by the SPL, the engineer can track the different impacts on the quality and single out optimization opportunities. Additionally, it is possible to determine if a feature selection breaks any given quality requirements, which is done by filtering all possible variants based on existing quality requirements.

V. EVALUATION

Evaluation criteria considered beyond the M and S requirements were an initial assessment of scalability for current SPL development including the usage of OCL for on-the-fly quality evaluation.

The measurements were performed on an Dell Latitude E6500 (Core2Duo CPU @ 2.53GHz) PC with 3.5GB RAM running Microsoft Windows XP Pro SP3, Java JDK 1.6, Eclipse Galileo 3.5 (Modeling Distribution SR1), openArchitectureWare 5, CVM framework 0.6.0, Eclipse OCL2.0 Interpreter 1.3, and the Eclipse Modeling Framework 2.5. All measurements were performed 3 times and averaged.

For the first set of measurements, the tailoring process (as shown in Figure 3) for binding the variability of the Quality Annotated Model to derive a single variant (Quality Model Instance) was considered to determine the current practical scalability limitations of variation points, features, and resulting generation time. Table I and Figure 19 show a nearly linear correlation between a change in the number of variation points and the generation time when the number of features was held constant. An increase in the number of features also showed a nearly linear increase in the generation time. This result is explained by the iterations used in the generator code implementation for each variation point and for each feature. As to conditions, varying the number of Boolean conjunctions up to 20 for a variation point made no significant difference due to other inherent overheads.

TABLE I. TAILORING PROCESS TIME (MSEC) GIVEN A NUMBER OF FEATURES AND VARIATION POINTS

Number of Variation Points	Total Number of Features		
	300	*600*	*900*
1500	4302	6411	8709
3000	6771	11307	15526
4500	9453	16016	22563

The derivation and quality analyses of all variants of a SPL can be computationally expensive and make its usage impractical. Thus, measurements on the performance of the implementation for automated variant derivation were performed to determine its limits. The time to derive all variants of a quality-annotated model with 100 QoSConstraints (simple additions) was measured for a binary structured feature tree of or-relations, while the number of features was increased from 10 to 32. The results in Table II and Figure 20 show that, given a current PC, the quality could be evaluated with up to 20 features and 2000 resulting variants on-the-fly with results in less than a minute.

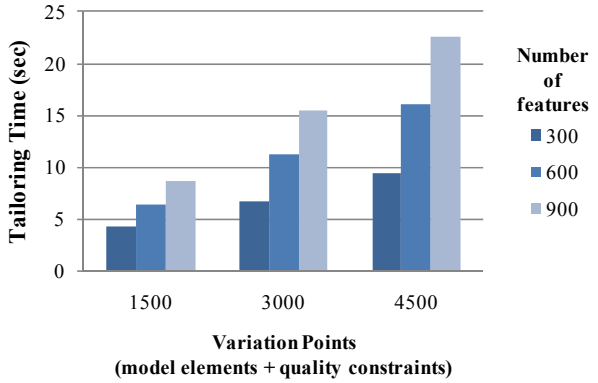

Figure 19. Tailoring process time (sec) vs. variation points and number of features.

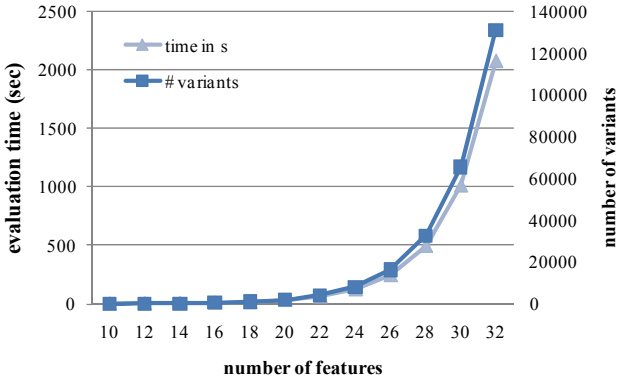

Figure 20. Quality evaluation time (sec) vs. number of features and variants.

TABLE II. QUALITY EVALUATION TIME GIVEN A NUMBER OF FEATURES AND VARIANTS

Features	Variants	Time (sec)
10	63	4
12	127	5
14	255	7
16	511	11
18	1023	18
20	2047	35
22	4098	65
24	8191	126
26	16383	247
28	32767	497
30	65535	1012
32	131071	2075

Based on these results with available tooling and systems, it is currently feasible to use and have the benefits of IQSPLE in industrial settings. The evaluation showed that performance for single variant quality evaluations was sufficient for usage today, but scalability issues were found

with handling large variation sets. When the quality evaluation of a large number of variants is desired, it is recommended that quality evaluations be executed in offline batch mode, and the results for all variants stored in a database for later access in order to enable tradeoff analysis to take place without encumbrances. Optimization possibilities include evaluating boundary constraints on the quality function properties to avoid further calculation overhead, e.g., aborting a calculation when a boundary value is exceeded.

VI. RELATED WORK

.Related work includes the Feature-Softgoal Interdependency Graph (F-SIG) approach [10], which supports quality modeling in the domain analysis phase. Its lack of support for quantitative values results in only imprecise quality assessments of a variant. The Extended Feature Model (Ext-FM) [11] applies a Constraint-Satisfaction-Problem approach and allows both quantitative and qualitative values to determine the set of matching variants. However, it requires a hierarchical modeling of quality attributes that restricts the possible set of quality dependencies that can be modeled. The Integrated Software Product Line Model (ISPLM) [12] integrates an implementation model that supports quantitative Q-attributes, yet it does not specify how these Q-attributes are to be utilized for a Q-assessment or set selection of variants. The Q-ImPrESS project [13] aims at modeling quality attributes at an architectural level. A reverse engineering process is used to derive component models which than are evaluated for quality prediction. It lacks support for modeling variation points in the problem space and in the solution space. Quality-driven Architecture Design and Analysis (QADA) [14] is a SPL architecture design method supporting traceable product quality and design-time quality assessment. Qualitative quality requirements are treated as architectural style(s) and patterns, and quantitative quality requirements as the properties of individual architectural elements. While addressing not only the conceptual architecture but also the concrete architecture, it does not produce implementation artifacts. It uses quality viewpoints [15] and conforms to OMG's Model-Driven Architecture (MDA) and IEEE 1471 [16] and uses UML profiles. [17] describes a tool chain that supports QADA including quality evaluation and test result imports. The protégé ontology tool is used for quality attribute definition, whereas IQSPLE encourages the use of feature tree tooling (e.g., CVM) for quality attributes due to its simplicity and prevalence. A comparison of these methodologies is shown in Table III with "Y" meaning yes and "N" meaning No.

COVAMOF [18] supports the modeling of dependencies between a set of variation points, however it does not explicitly address quality modeling.

While the Attribute Driven Design (ADD) method [19] supports the explicit articulation of the quality attribute goals for SPLs, it is narrowly focused on the definition of the conceptual architecture.

With regard to addressing SPL variability and quality annotation in UML models, the comparison matrix in Table IV shows an assessment of related SPLE approaches for a subset of requirements. 'Quality annotation' refers to the capability of annotating existing artifacts in SPL with quality information. 'Requirements analysis' refers to the support of the requirements process from elicitation to documentation, while 'feature model integration' means the usage of feature models as a basis for the approach. 'Negative variability' and 'structural variability' are defined in [2] and describe the means to define SPL artifacts and transfer them into product instance artifacts. 'UML compliant' refers to the restriction of using standardized modeling based on UML including OCL, stereotypes, tagged values, etc., an influential factor for industrial adoption of an approach. Modeling artifacts can contain 'modeling constraints' (i.e. constraints defined in the solution space) and 'configuration constraints' (i.e. interdependencies of features in feature trees). Constraints might become expensive to evaluate, thus a separation of constraints that can be evaluated on-the-fly and constraints that will only be checked during the generation process might become necessary, evaluated under 'checks during generation'. 'Product instantiation' evaluates the ability to create a definition of the derived product instance, the simplest form of which could be a list of selected features. An approach can explicitly include 'code generation' in its process and allow 'quality variability tracing across elements', meaning selections and bindings through the artifacts.

TABLE III. METHODOLOGY COMPARISON FOR QUALITY SUPPORT

Requirement	F-SIG	Ext-FM	ISPLM	Q-ImPrESS	QADA	IQSPLE
M1: qualitative values	Y	Y	N	Y	Y	Y
M2: quantitative values	N	Y	Y	Y	Y	Y
M3: algorithms for calculating the resulting attribute values	N	Y	N	Y	N	Y
M4: presentation as feature	N	N	N	N	N	Y
M5: artifact annotation	N	N	N	Y	Y	Y
S1: calculate the quality values of a given variant	N	Y	-	Y	Y	Y
S2: determine the set of possible variants	N	Y	-	N	N	Y
S3: constrain the selectable features	N	N	-	N	N	Y
S4: visualization of quality values	N	N	N	Y	N	Y
S5: quality modeling integration in solution space	N	N	N	Y	Y	Y
S6: reuse of artifacts	N	N	Y	Y	Y	Y
S7: traceability support	N	N	Y	Y	Y	Y

TABLE IV. METHODOLOGY COMPARISON FOR UML VARIABILITY SUPPORT

	SPLIT	PLUS	MDD-AO-PLE	UML ext. [22]	IQSPLE
quality annotation					✔
requirement analysis	+++	+++	++	+	++
feature model integration		✔	✔	✔	✔
negative variability				✔	✔
structural variability	✔	✔		✔	✔
UML compliant	✔	✔		✔	✔
modeling constraints	✔	✔	✔	✔	✔
configuration constraints			✔	✔	✔
checks during generation			✔	✔	✔
product instantiation	+++	+	+++	+	+
code generation			✔		✔
quality variability tracing across elements					✔

Approaches include the conceptual framework SPLIT [20], where additional UML stereotypes, e.g., <<variabilityMechanism>> and <<variationPoint>>, are used for specifying variable elements. However, SPLIT does not integrate an abstract feature view as does the IQSPLE, and the variation points and the corresponding variants require a separate class that may cause transparency issues in large SPLs. PLUS (Product Line UML-Based Software Engineering) [21] extends UML to model variability and commonality using stereotypes and primarily subclassing. [22] presents a generic modeling approach with additional variability stereotypes as extensions to UML. The variation points and variants can be assigned with tagged values to define certain properties, such as the binding time of variants, the multiplicity of associable variants, and the condition of binding. However, this approach does not address the derivation of product line instances. Crosscutting variability in SPLs is investigated in MDD-AO-PLE [23][24][25] and related aspect-oriented (AO) SPLE work. While this work has not specifically addressed the difficulties described in this paper for quality modeling integration, the combination of these AO techniques with IQSPLE could be synergistic and should be investigated in future work.

VII. CONCLUSION AND FUTURE WORK

By integrating quality modeling into SPLs with a holistic method and tool approach, the application of IQSPLE results in product as well as process benefits. At the product level, the resulting product instance has a higher potential value to the customer since it is more likely to conform to his or her expectations, satisfying not only requested features but also complying with quality expectations. At the process level, the use of an interactive, quality-driven selection process provides efficiencies by supporting on-the-fly evaluation and visualization for expeditious trade-off analysis while assisting with and automatically constraining valid variant selection against quality requirements. Efficiency is also furthered via reuse of quality modeling and annotations throughout the SPL lifecycle. The effectiveness of the process is enhanced by making qualities first-class requirement entities that are analyzed and formalized. The comprehensive approach promotes contemplation of (aggregated) quality impacts from the initial SPL concept since qualities can be (directly) annotated across the solution space artifacts (and are available directly to SPL engineers). SPL process effectiveness is also furthered by automatic evaluation and optimization as well as traceability support for the source of variability decisions. IQSPLE supports the flexible derivation of individual variants instead of a limited set of predefined variants via immediate feedback on the resulting quality attribute values of the current selection. Barriers to adoptability of quality modeling in industrial SPLs are lowered by IQSPLE due to its avoidance of unnecessary complexity (e.g., ontologies are avoided) and focus on integrating known and common tooling and standards (UML, OCL, stereotypes, tagged values, OMG's QoS profile, MDD (e.g., oAW), Eclipse, feature trees).

By fulfilling the M and S requirements, IQSPLE supports qualities in quantitative and qualitative ways. The application of constraints on features allows the explicit modeling of quality values inside a feature tree. It is not necessary to make any structural changes to feature trees or add any additional implementation details, so feature trees can still be used for customer communications. With quality functions, a mechanism is provided to transform different quality impacts into a single quality characteristic and thus make it possible to compute qualities of a variant. To make it easier to recognize how changes in feature selections affect qualities, quality values can be displayed in spider charts.

The eHealth scenario shows how IQSPLE supports the detection of optimization opportunities and trade-offs during product instantiation, making use of the traceability of the modeled dependencies.

Models are necessarily limited in their portrayal of reality, and holistic quality modeling of a SPL faces significant challenges due to the large set of variations. While IQSPLE may contribute towards improved quality modeling in SPLs, much work remains. Future work should address visualization of influences on qualities via quality interaction dependency graphs; model checking integration (e.g., UML dependency changes may affect OCL constraints); OCL validity and syntax-checking of, e.g., component names and quality attributes, perhaps eventually code-completion support; the support for and integration of developmental qualities (e.g., architectural metrics affected by a specific configuration); replacing the current brute-force variant evaluation algorithm by alternatives described in "Evaluation" Section V; decision support via analysis of potential variants; enabling tolerances and trade-offs in formulas for automatic optimization; and the application of the IQSPLE in other industrial domains beyond the medical domain.

REFERENCES

[1] J. Bartholdt, M. Medak, and R. Oberhauser, “Integrating Quality Modeling with Feature Modeling in Software Product Lines”, Proc. of the Fourth International Conference on Software Engineering Advances (ICSEA 2009), IEEE Computer Society, 2009.

[2] J. Bartholdt, R. Oberhauser, A. Rytina, „Addressing Data Model Variability and Data Integration within Software Product Lines”, International Journal On Advances in Software, ISSN 1942-2628, vol. 2, no. 1, 2009, pp. 86-102

[3] F.J. v.d. Linden, K. Schmid, and E. Rommes, “Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.” Springer, 2007, ISBN 3540714367.

[4] K. Pohl, G. Böckle, and F.J. v.d. Linden, “Software Product Line Engineering: Foundations, Principles and Techniques”. Springer, 2005, ISBN 3540243720.

[5] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute, Carnegie Mellon University, 1990.

[6] M. L. Griss, J. Favaro, and M. d. Alessandro, “Integrating feature modeling with the RSEB,” Proc. of the 5th International Conference on Software Reuse (1998), ICSR, IEEE Computer Society.

[7] J. Bayer, et al “PuLSE: a methodology to develop software product lines”, In Proceedings of the 1999 Symposium on Software Reusability (SSR '99), ACM, pp. 122-131, 1999.

[8] I. Ozkaya, L. Bass, R. S. Sangwan, and R. L. Nord, “Making Practical Use of Quality Attribute Information,” IEEE Softw. 25, 2 (Mar. 2008), pp. 25-33. DOI= http://dx.doi.org/10.1109/MS.2008.39.

[9] OMG, UML Profile for Modeling Quality of Service & Fault Tolerance Characteristics & Mechanisms, v1.1, formal/08-04-05

[10] S. Jarzabek, B. Yang, and S. Yoeun, “Addressing quality attributes in domain analysis for product lines,” IEE Proceedings Software, vol. 153, no. 2, pp. 61–73, 2006.

[11] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, “Automated reasoning on feature models”, in LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005, pp. 491–503, 2005.

[12] N. Siegmund, M. Kuhlemann, M. Rosenmüller, C. Kästner, and G. Saake, “Integrated product line model for semi-automated product derivation. Using non-functional properties,” Proc. of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS), pp. 25–31, 2008.

[13] S. Becker, M. Hauck, M. Trifu, K. Krogmann, J. Kofron, “Reverse Engineering Component Models for Quality Predictions”, CSMR 2010, IEEE, Mar 2010.

[14] M. Matinlassi, E. Niemelä, and L. Dobrica, "Quality-driven architecture design and quality analysis method, A revolutionary initiation approach to a product line architecture," VTT Technical Research Centre of Finland, Espoo, 2002.

[15] A. Purhonen, E. Niemelä, and M. Matinlassi, "Viewpoints of DSP Software and Service Architectures," Journal of Systems and Software, vol. 69, 2004, pp. 57 - 73.

[16] IEEE, "IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems," Std-1471-2000. New York: Institute of Electrical and Electronics Engineers Inc., 2000.

[17] A. Evesti, E. Niemel, K. Henttonen, M. Palviainen, "A Tool Chain for Quality-Driven Software Architecting," splc, pp.360, 2008 12th International Software Product Line Conference, 2008.

[18] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, “Modeling dependencies in product families with COVAMOF”, Proc. of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2006), March 2006.

[19] F. Bachmann and L. Bass, "Introduction to the Attribute Driven Design Method," icse, pp.0745, 23rd International Conference on Software Engineering (ICSE'01), 2001.

[20] M. Coriat, J. Jourdan, and F. Boisbourdin, “The SPLIT method: building product lines for software-intensive systems,” In Proceedings of the First Conference on Software Product Lines: Experience and Research Directions (Denver, Colorado, United States). P. Donohoe, Ed. Kluwer Academic Publishers, Norwell, MA, 2000, pp. 147-166.

[21] H. Gomaa, “Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures”, Addison-Wesley, 2005, ISBN 0201775956.

[22] M. Clauss, “Generic modeling using UML extensions for variability”, In Proceedings of the Workshop on Domain Specific Visual Languages, OOPSLA 2001, Jyväskylä University Printing House, Jyväskylä, Finland, 2001, ISBN 951-39-1056-3, pp. 11-18.

[23] M. Voelter and I. Groher, “Handling Variability in Model Transformations and Generators”, in Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM’07), Sprinkle, J., Gray, J., Rossi, M., Tolvanen, J.-P., (eds.), Computer Science and Information System Reports, Technical Reports, TR-38, University of Jyväskylä, Finland 2007, ISBN 978-951-39-2915-2.

[24] M. Voelter and I. Groher, “Product Line Implementation using Aspect-Oriented and Model-Driven Software Development,” In Proceedings of the 11th international Software Product Line Conference (September 10 - 14, 2007). International Conference on Software Product Line. IEEE Computer Society, Washington, DC, 2007, pp. 233-242.

[25] I. Groher, "Aspect-Oriented Feature Definitions in Model-Driven Product Line Engineering", Dissertation, Johannes Kepler Universität, Linz, April 2008.