Integrating Quality Models and Static Analysis for Comprehensive Quality Assessment

Klaus Lochmann Lars Heinemann
Technische Universität München, Garching b. München, Germany
{lochmann,heineman}@in.tum.de

ABSTRACT

To assess the quality of software, two ingredients are available today: (1) quality models defining abstract quality characteristics and (2) code analysis tools providing a large variety of metrics. However, there exists a gap between these two worlds. The quality attributes defined in quality models are too abstract to be operationalized. On the other side, the aggregation of the results of static code analysis tools remains a challenge. We address these problems by defining a quality model based on an explicit meta-model. It allows to operationalize quality models by defining how metrics calculated by tools are aggregated. Furthermore, we propose a new approach for normalizing the results of rule-based code analysis tools, which uses the information on the structure of the source code in the quality model. We evaluate the quality model by providing tool support for both developing quality models and conducting automatic quality assessments. Our results indicate that large quality models can be built based on our meta-model. The automatic assessment shows a high correlation between the automatic assessment and an expert-based ranking.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Product metrics*; D.2.9 [**Software Engineering**]: Management—*Software Quality Assurance (SQA)*

General Terms

Measurement

Keywords

Quality Models, Software Metrics, Static Code Analysis

1. INTRODUCTION

Motivation. To effectively manage software costs it is essential to assess the quality of software. For this, two major ingredients are available today.

(1) Code analysis tools provide a large range of metrics that can be used as indicators for the quality of a software product. However, these metrics focus on very special aspects of the source code and it is therefore difficult to use them for obtaining a comprehensive overview of the overall quality of a software system.

(2) Quality models like ISO 25010 [2] and others [3, 7, 6] define high-level quality attributes, which are commonly used to characterize the quality of software. However, these quality models are too abstract to be operationalized for the quality assessment of a software system.

Problem. There exists a gap between abstract quality characteristics defined in quality models on the one hand and isolated quality analysis tools on the other hand. The problem of obtaining quality assessments using static code analysis can be split into two parts:

(1) The aggregation of the results of heterogenous static code analysis tools to an overall quality assessment of a software product remains a challenge. For tools that address this challenge, adequate user assistance is required for defining meaningful and comprehensible aggregation specifications. Especially for rule-based static code analysis tools that produce rule violation messages (like FindBugs[1] and PMD[2]) associated with code locations (called by us *findings*) rather than metric numbers, little work exists on how to aggregate them. Besides the simplistic *defect density* (number of findings per lines of code) approaches exist that allow to specify arbitrary mathematical expressions to aggregate the results. There is no systematic approach for obtaining meaningful and comprehensible aggregation specifications for rule-based static code analysis tools.

(2) An additional challenge is the large number of rules provided by static code analysis tools. Organizing them and working with them is not possible without a mechanism to structure and classify them in a comprehensible manner.

Contribution. In this paper we present a quality model that addresses both problems. An explicit meta-model defines a comprehensive structure for quality models. This structure allows us to build adequate tool-support for managing quality models growing very large.

Regarding the problem of aggregating the results of rule-based code analysis tools, we propose a new approach for normalizing them. This approach is tightly coupled to the quality model. The quality model provides the necessary background information on the structure of the source code, in order to enable the user to define meaningful aggregations.

[1]`http://findbugs.sourceforge.net`
[2]`http://pmd.sourceforge.net`

We evaluate the approach by providing tool support for both developing quality models and for conducting automatic quality assessments. A case study shows the suitability of the quality model to cover a large number of rules from two rule-based static analysis tools. To evaluate the meaningfulness of the produced quality assessments we apply the automatic assessment based on the quality model to 17 open source systems. Our results are analyzed for their diversification and for their consistency with an independent expert-based assessment. The results indicate that the diversification is satisfactory and that there is a high correlation between the two assessment results.

2. RELATED WORK

The related work can be split into three parts:
(1) *Quality models* were proposed to structure and understand the multi-faceted concept of software quality [3, 5, 6]. These models define quality by decomposing it into more manageable quality attributes. However, these quality attributes are too abstract to be directly measurable and do not allow for an operationalization in terms of a quality assessment of real software systems. In contrast, our model allows to define quality from abstract characteristics down to concrete measures.
(2) *Code analysis tools* determine software metrics and quality defects like coding guideline violations or bug patterns (*e. g.*, FindBugs[1], PMD[2], Checkstyle[3]). Although these tools are highly valuable within their specific scope, it remains unclear how to deduct from the results of them a comprehensive assessment of the overall quality of a software system.
(3) *Dashboard tools* integrate multiple quality analyses and allow to aggregate and visualize their results in a comprehensive quality dashboard (*e. g.*, Sonar[4], XRadar[5], QALab[6]). However, in these tools the metrics are either not explicitly linked to a quality model or they use a fixed quality model and thus do not allow for project-specific definition of software quality.

The approach of Schackmann et al. [9] connects measurement tools with a quality model. The quality model is defined in their tool, which requires the user to define the functions for aggregating the measurement values. However, their approach allows arbitrary functions, and gives therefore no help to the user in specifying meaningful ones.

The research project Squale[7] develops a quality model based on existing standards (*e. g.*, ISO 9126) and tool support that allows to aggregate metrics from third party analysis tools to high level quality factors. While Squale uses a fixed quality model, our approach allows for a project-specific modeling of quality. Moreover, Squale integrates with a fixed set of analysis tools. In contrast, our approach allows the flexible integration of analysis tools. In addition we can also integrate the results of manual quality assessments like code inspections or reviews.

[3]`http://checkstyle.sourceforge.net`
[4]`http://www.sonarsource.org`
[5]`http://xradar.sourceforge.net`
[6]`http://qalab.sourceforge.net`
[7]`http://www.squale.org`

3. QUALITY MODEL

The quality model pursuits three goals: (1) to structure rules of static analysis tools in a comprehensible way, (2) to define their influence on quality characteristics of the software product, and (3) to define comprehensible and understandable aggregation formulas for automatically calculating quality assessments of a software product.

The quality meta-model presented here is a variant of the model developed in the project *Quamoco*[8]. This variant aims for clear and understandable aggregation specifications.

3.1 Structure

The quality model relies on a product model of software, which describes the entities a software product consists of. The idea of using a product model as a backbone of the quality model has been proposed in similar forms in literature [5, 4]. In our quality model, we model the parts of the software product as **Entities**. Between entities we define a specialization relation **is-a** and a decomposition relation **part-of**. Since we model the rules of static code checkers, the entities usually refer to the source code of the software. Typical entities include *Class* and *Expression*, whereby Expression is further refined by *Relational expression* and *Arithmetical expression*, which are in an *is-a* relation to Expression.

The main concept of the quality model is a **Property**. As the name suggests, it describes a property of the software product. In the literature, the term property is defined as an attribute which is used to characterize an object. The notion of an object that is characterized is central to all definitions. In our quality model, we define a **Property** as an **Attribute** of an **Entity**. In the following we use the notation [*attribute*@*entity*] for properties.

In our quality model, we use entities that characterize the entire product, to express the "-ilities" of the ISO 25010 [2]; for example the "maintainability" is expressed as [Maintainability@Product]. Other attributes are used for parts of the product, for example "Correctness", resulting in [Correctness@Arithmetical expression].

Properties can be refined, using the **Refines** relation. If a property B refines a property A, it means that B is more special regarding the entity it characterizes than A. More precisely, if a property B refines a property A, then the entity of B must be in a *is-a* relation to the entity of A. For A= $[a@e_1]$ and B= $[a@e_2]$, written formally that is:

$$[a@e_1] \xleftarrow{\text{ref}} [a@e_2] \Rightarrow e_1 \xleftarrow{\text{is-a}} e_2 \qquad (1)$$

The refinement relation is used to create a hierarchy of properties according to the inheritance hierarchy of the entities. The entity hierarchy, can be used as a reference for the refinement of multiple properties.

Figure 1 illustrates an excerpt of the quality model. An example for a refinement relation is the property [*Correctness*@*Expression*] that is refined by [*Correctness*@*Boolean Expression*] and [*Correctness*@*Arithmetical Expression*].

A property can be **decomposed** into sub-properties by specializing its attribute, while the entity keeps the same. Formally, the properties of this relation have always the following form:

$$[a_1@e] \xleftarrow{\text{dec}} [a_2@e] \qquad (2)$$

[8]`http://www.quamoco.de`

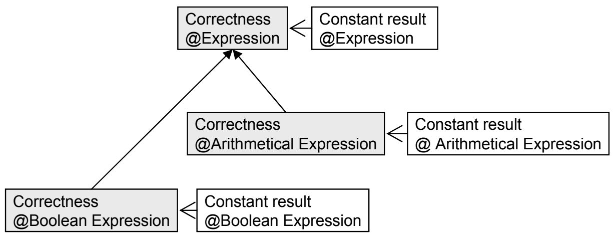

Correctness @Expression
Constant result @Expression
Correctness @Arithmetical Expression
Constant result @ Arithmetical Expression
Correctness @Boolean Expression
Constant result @Boolean Expression

Figure 2: Example: inheritance of properties

Figure 1 contains an excerpt from the quality model, showing an example for the decomposition relation. The property $[\mathit{Quality@Product}]$ is decomposed into $[\mathit{Functional\ Suitability@Product}]$ and $[\mathit{Maintainability@Product}]$.

With the refinement relation, we can define a property for a rather abstract entity, and then "inherit" the property to the sub-entities. The same principle is applied to the decomposition relation: the decomposing properties are **inherited** alongside the refinement relation. Formally, this means:

$$[a_1@e_1] \xleftarrow{\text{dec}} [a_2@e_1] \land [a_1@e_1] \xleftarrow{\text{ref}} [a_1@e_2] \qquad (3)$$
$$\Rightarrow [a_1@e_2] \xleftarrow{\text{dec}} [a_2@e_2]$$

Figure 2 shows an example. The property $[\mathit{Correctness@Expression}]$ is decomposed by $[\mathit{Constant\ result@Expression}]$. This expresses that for all (complex) expressions in source code it is suspicious if they yield a constant result. Then, $[\mathit{Correctness@Expression}]$ is refined by $[\mathit{Correctness@Boolean\ expression}]$ and $[\mathit{Correctness@Arithmetic\ expression}]$. Both these properties inherit the *Constant result* attribute and must specify what *Constant results* means for them and a method for measuring it.

While the refinement and decomposition relations describe a breakdown of properties, the **Impact** relation describes a qualitative interrelation between properties. More precisely, an impact specifies that the degree to which an entity has a property influences the degree of which another entity has another property. The effect of the influence can be either positive or negative. If the impact has a positive effect, the degree of which the entity has the target property is increased if the entity has the source property; and vice-versa for a negative impact. Because the knowledge on impacts is usually present in a qualitative manner, a rationale for each impact is given in prose. An impact of a property A on a property B may only exist if they either characterize the same entity or if the entity of A is in a *part-of* relation to the entity of B. Formally, we define:

$$[a_1@e_1] \overset{\text{imp}}{\rightsquigarrow} [a_2@e_2] \Rightarrow e_1 \xleftarrow{\text{part-of}} e_2 \qquad (4)$$

The excerpt from the quality model (see Figure 1) illustrates an example of an impact. The property $[\mathit{Correctness@Arithmetical\ expression}]$ has an impact on $[\mathit{Functional\ Correctness@Product}]$.

Through the two relations *refines* and *decomposes* different trees of properties are defined, which we call *property trees.* The *impact* relation describes how properties of one property tree influence properties of another property tree. Typically there are two property trees in a quality model (see Figure 1): (1) the tree on the left describes properties characterizing *Product part*s. They get their values from metrics of static code analysis tools. (2) the tree on the right describes properties characterizing *Product.* These properties are impacted by the properties of the first tree. The values of these properties are calculated through the impacts.

3.2 Quality Assessment

Conducting a quality assessment using the quality model means determining the degree to which a property is present in an entity. We define that this degree is expressed by either a range of real numbers between 0 and 1, $\mathbb{D} = [0,1]$, whereby 0 means that the property is not present and 1 means that the property is fully present, or by boolean values $\mathbb{B} = \{0,1\}$. Since an *entity* of the quality model denotes a class of objects, a concrete evaluation is conducted on an instance of an entity. This evaluation is denoted as a function

$$\mathit{eval} : \mathit{Property} \times \mathit{Object} \to \mathbb{D}$$

whereby *Property* is the set of all properties and *Object* is the set of all objects. For example if the property is $[\mathit{Correctness@Class}]$, then an evaluation of this property is conducted for a concrete class with the name "MyClass1":

$$\mathit{eval}([\mathit{Correctness@Class}], \text{"MyClass1"}) = 0.3$$

In order to calculate the evaluation function in a concrete quality model, actual determination functions must be defined. There are principally two ways of doing so: (1) direct determinations and (2) indirect determinations.

3.2.1 Direct Determination

The value of a property can be directly determined, by specifying a determination function that directly determines its value. This is usually done for the leaf-properties in a property tree (see Figure 1). Direct determination functions are called **instrument**. An instrument can be a tool that calculates some metric on the source code. There are two principally different types of results produced by these tools:

1. **Metrics:** We define metrics as tools that calculate the *eval*-function using the full value ranges: $\mathit{Property} \times \mathit{Object} \to \mathbb{D}$. A typical example for such a metric is the cyclomatic complexity, which is calculated for each method. It can be directly used to evaluate $[\mathit{Complexity@Method}]$.
2. **Findings:** We define findings as results of rule-based static code analysis tools. These tools report locations in the source code where a rule is violated. For such tools we define:
   $\mathit{eval}(p, \mathit{obj}) = 1$ if there is a finding for *obj*
   $\mathit{eval}(p, \mathit{obj}) = 0$ if there is no finding for *obj*
   This way, all properties relying on data of tools producing findings, are evaluated on a boolean scale: $\mathit{eval} : \mathit{Property} \times \mathit{Object} \to \mathbb{B}$.

For metrics a large number of methods for aggregating them are known in the literature. They can be aggregated by using for example *mean*, *median*, etc. The aggregation of findings is much more challenging and a main topic of this paper.

3.2.2 Indirect Determination: Refinement

Indirect determination functions determine the value of a property by relying on the values of the properties refining, decomposing, and impacting it. These are all inner nodes in the properties trees (see Figure 1). In this case the evalua-

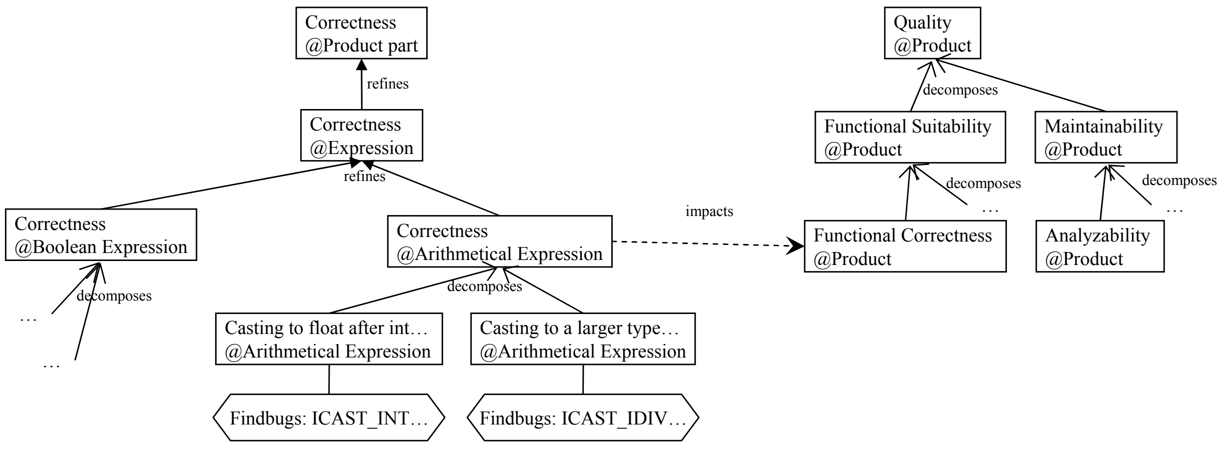

Correctness @Product part
refines
Correctness @Expression
refines
Correctness @Boolean Expression
decomposes
…
…
Correctness @Arithmetical Expression
decomposes
Casting to float after int… @Arithmetical Expression
Casting to a larger type… @Arithmetical Expression
Findbugs: ICAST_INT…
Findbugs: ICAST_IDIV…
impacts
Quality @Product
decomposes
Functional Suitability @Product
Maintainability @Product
decomposes
…
decomposes
…
Functional Correctness @Product
Analyzability @Product

Figure 1: Excerpt of the Quality Model

tion function must take care of aggregating and normalizing the values of its sub-properties.

If a property p is refined by other properties p_i, $i = 1..n$, the attributes of all these properties are the same, while the entities of the sub-properties p_i are a specialization of the entity of p. When evaluating an object regarding p, it's evaluation function "chooses" the property p_i so that the object is an instance of its entity. Let $p = [a@e], p_i = [a@e_i]$ then the evaluation function is defined as follows:

$$eval([a@e], o) = \begin{cases} eval([a@e_1], o) & \text{if } o \in e_1 \\ \dots \\ eval([a@e_n], o) & \text{if } o \in e_n \end{cases}$$

3.2.3 Indirect Determination: Decomposition

If a property p does not have refinements, but is decomposed by properties $p_1, \dots, p_n$, all these properties characterize the same entity as p, and have different attributes. Consequently, all sub-properties can be evaluated regarding the object under consideration. The evaluation of p then results from an aggregation of the results of the sub-properties. Written formally, this means:

$$eval(p, obj) = aggr(eval(p_1, obj), \dots, eval(p_n, obj))$$

The aggregation function $aggr$ must be provided by each property that is decomposed by other properties. There are two cases:

1. **Findings:** If all sub-properties are defined by findings, than all sub-properties are evaluated on a boolean scale. Therefore, also the $aggr$-function is defined on a boolean scale: $aggr : \mathbb{B}^n \to \mathbb{B}$. This means that as aggregation function, boolean functions can be specified. Our experience in creating quality models has shown that there are four standard aggregation functions that are sufficient:
   (a) $aggr(v_1, \dots, v_n) = v_1 \wedge \dots \wedge v_n$
   (b) $aggr(v_1, \dots, v_n) = v_1 \vee \dots \vee v_n$
   (c) $aggr(v_1, \dots, v_n) = \neg v_1 \wedge \dots \wedge \neg v_n$
   (d) $aggr(v_1, \dots, v_n) = \neg v_1 \vee \dots \vee \neg v_n$
   The first function expresses for example that a property p is satisfied if all its sub-properties are satisfied; while the third function expresses that p is satisfied if none of its sub-properties is satisfied. The first function is used for example, to express that [*Correctness@Expression*] is satisfied if both [*Correctness@Boolean Expression*] and [*Correctness@Arithmetical Expression*] are satisfied.
2. **Properties on $\mathbb{D}$:** If one of the sub-properties is defined by a metric or by incoming impacts, than it is evaluated on the scale $\mathbb{D} = [0, 1]$. In this case, one of the following functions are usually applied: $aggr(v_1, \dots, v_n) = \text{mean/median/}\dots(v_1, \dots, v_n)$.

3.2.4 Indirect Determination: Impact

A third possibility to calculate the evaluation of a property is to use the properties with an impact on that property. When an impact is evaluated in this way the issue of **normalization** arises. An example will demonstrate the challenge behind it. Let's consider an impact $[Appropriateness@Class] \overset{\text{imp-}}{\rightsquigarrow} [Analyzability@Product]$. A product that is evaluated may consist of multiple classes. Therefore, the values of the single classes have to be combined into one value for the product. To complicate matters further, the different sizes of the classes have to be taken into account, to come up with reasonable results. Consider, for example, a product that consists of six classes; one class contains 90% of the source code, and the five remaining classes account for 10%. In this case the appropriateness of the large class must obviously have a greater influence than that of the small classes.

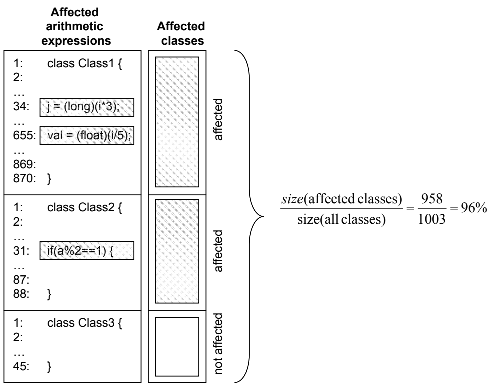
To calculate such impacts in the quality model we draw back on the simplifying assumption that the properties of product parts are evaluated on a boolean scale. This way, an object may get a 1 for a property (we call the object *not affected* by the property), or the object may get a 0 for a property (we call the object *affected* by the property). The idea of determining a value for the impact is then to calculate the proportion of the size of affected parts of the system in comparison to not affected parts.

Example 1. The impact $[Appropriateness@Class] \overset{\text{imp}}{\rightsquigarrow} [Analyzability@Product]$ is evaluated. First, for all objects of type "Class" the property is calculated. That means, each class is either "affected" or "not affected". Next, the size of each class is calculated, and the evaluation of the impact is the proportion of the sizes of affected classes versus unaffected classes: $size\,(\text{affected classes})\,/size\,(\text{all classes})$.

This principle of calculating the size of affected vs. not affected parts of the system works, if meaningful measures for the size of the entity under consideration can be found. For classes and methods, we use the lines of code as a measure of their size. We cannot apply this principle to other entities directly. For example, it does not make sense to calculate the size in lines of code of the entity *Arithmetical expression*.

Therefore, we extend our principle to so called **Ranges**. If we want to calculate the impact of a property, whose entity has no meaningful size, we define one of its super entities (regarding the part-of relation) as its *range*. The entity we choose as range, must have a meaningful size. Then, we define that an object of the range-entity is affected, if it contains at least one object that is affected. This way, for each range-object it is defined whether it is affected or not, and the principle of the normalization can be applied to the range object.

Example 2. The impact $[\mathit{Correctness@Arithmetical\ expression}] \overset{\text{imp}+}{\rightsquigarrow} [\mathit{Functional\ correctness@Product}]$ is calculated. *Arithmetical expression* is part of *Method*, which is part of *Class*. We choose *Class* as range of this impact. This means, a class is *affected* if it contains at least one arithmetical expression that is not correct; and it is not affected if it contains no such arithmetical expression. Then, the principle of normalization is applied, and the size of affected classes is set in proportion to the size of all classes. Figure 3 illustrates this.



```
1:    class Class1 {
2:
...
34:   j = (long)(i*3);
...
655:  val = (float)(i/5);
...
869:
870: }

1:    class Class2 {
2:
...
31:   if(a%2==1) {
...
87:
88:  }

1:    class Class3 {
2:
...
45:  }
```

$$\frac{size(\text{affected classes})}{\text{size(all classes)}} = \frac{958}{1003} = 96\%$$

Figure 3: Example: Normalization

For a given property $[a@e]$, all applicable ranges can be determined using the part-of relation of entities. All entities e_i of which e is part-of are potential ranges, if they support the determination of their size. Consider the following example of an entity hierarchy:

$$\begin{tikzcd}
 & & \text{Field} \arrow[dl] & \\
\text{Source file} & \text{Class} \arrow[l] & & \\
 & & \text{Method} \arrow[ul] & \text{Expression} \arrow[l]
\end{tikzcd}$$

Here, for expressions, the ranges *Source file*, *Class*, and *Method* are applicable, while for fields only *Class* and *Source file* are applicable. This way, the entity-hierarchy prescribes which ranges are applicable and it is automatically ensured that only actually calculable ranges are used.

Up to now, we have only described how one impact $p_i \overset{\text{imp}}{\rightsquigarrow} p$ is calculated. In real models, however, a property p is usually impacted by multiple properties p_i. Therefore, the evaluation values of all impacts $eval(p_i \overset{\text{imp}}{\rightsquigarrow} p), i = 1..n$ have to be aggregated. To allow an adjustable aggregation, we first apply a 3-point linear distribution to the value of each impact and than we calculate the mean of all values.

$$eval(p, obj) = \frac{1}{n} \sum_{i=1..n} linearDist\left(eval\left(p_i \overset{\text{imp}}{\rightsquigarrow} p\right), b_1, b_2, b_3\right)$$

b_1, b_2, b_3 are thresholds for the distribution function and have to be provided for each impact in the quality model.

4. TOOL SUPPORT

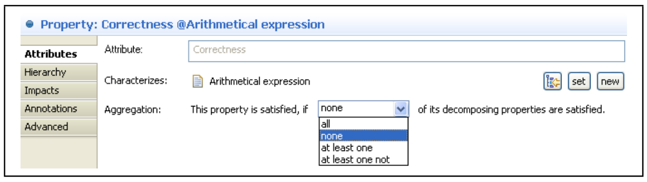
Based on the tooling of *Quamoco* we developed tool-support for both editing quality models and conducting automatic quality assessments based on a quality model.

4.1 Quality Model Editor

The quality model editor and a quality model for Java are available online[9].

The main view of the quality model editor is a tree of properties, defined by the relations *decomposition* and *refinement*. This tree allows adding and deleting properties. When a new property is added to this hierarchy, the user has to decide if it is refining or decomposing its parent-property. The editor automatically sets the attribute or the entity of the property correctly, i.e. a refining property must have the same attribute as its parent-property (formula 1), and a decomposing property must have the same entity as its parent property (formula 2). Furthermore, when changing the attribute or entity of a property, the editor only allows to create valid models:
(1) That means, if a property is refined by another property, its attribute cannot be changed (in the screenshot in Figure 4 the field *attribute* is disabled); and if the attribute of a property is changed, than the attributes of all refining properties are changed automatically.
(2) For example, if the entity of a refining property is changed, only sub-entities of the entity of its parent property are shown for selection (satisfying formula 1). Furthermore, the editor checks the constraint for inheritance (formula 3) and shows a warning if a property is not correctly inherited alongside the refinement hierarchy.

If a property is decomposed by other properties, it is necessary to specify an aggregation function (see Section 3.2.3). The four aggregation functions for boolean properties that were identified as commonly used are integrated into the editor. They can be specified using sentence patterns as shown in the screenshot in Figure 4.



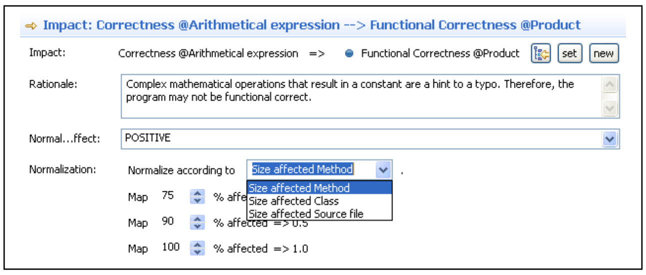
Attribute: Correctness
Characterizes: Arithmetical expression
Aggregation: This property is satisfied, if none of its decomposing properties are satisfied.
all
none
at least one
at least one not

Figure 4: Screenshot: Editing a property that is decomposed into other properties

For each impact in the model, it must be specified how it is normalized. The concept of ranges (see Section 3.2.4) is fully integrated into the editor. Depending on the entity characterizing the property under consideration, a list of all possible ranges is determined. This takes the part-of relation

[9]http://www4.in.tum.de/~lochmann/WETSoM2011/

of the entity and the available measures of the tooling into consideration. Furthermore, the linear distribution function and its thresholds can be edited in a table.
Figure 5 shows a screenshot of the form, whereby the impact $[Correctness@Arithmetical\ expression] \overset{\text{imp}+}{\rightsquigarrow} [Functional\ correctness@Product]$ is being edited. The user can choose between three ranges: *method*, *class*, and *source file*. For the linear distribution function, the user can enter the threshold values. In the example the following mapping was entered: 75% affected $\mapsto$ 0.0, 90% affected $\mapsto$ 0.5, and 100% affected $\mapsto$ 1.0.



Impact: Correctness @Arithmetical expression --> Functional Correctness @Product

Impact: Correctness @Arithmetical expression => Functional Correctness @Product

Rationale: Complex mathematical operations that result in a constant are a hint to a typo. Therefore, the program may not be functional correct.

Normal...ffect: POSITIVE

Normalization: Normalize according to Size affected Method .

Map 75 % affected => 0.0

Map 90 % affected => 0.5

Map 100 % affected => 1.0

Figure 5: Screenshot: Editing an impact

4.2 Automatic Quality Assessment

For conducting quality assessments with the quality model, based on *ConQAT*[10] and the tool support of *Quamoco*, an assessment tool was developed. ConQAT is an open source framework for building quality dashboards, by integrating external analysis tools. For reasons of brevity, a detailed description of this tooling cannot be provided here. However, the output of the assessment tool for the systems used in the case study is available online[9].

5. CASE STUDY

We perform a case study to show the feasibility of our approach. Using the quality model editor, we model rules of static code checkers in a quality model. Then, we perform an automatic quality assessment of software systems.

5.1 Design and Procedure

Based on the problem we intent to solve, we structure the case study through two research questions.

RQ1: Is the Proposed Quality Meta-Model Suitable to Model a Large Number of Static Code Analysis Rules?

We assess, how the structure of the meta-model enables us to create elaborate tool-support and how it supports the user in creating quality models. Then, we construct a quality model with a large number of static code analysis rules and discuss whether it is clearly structured.

RQ2: Does the Quality Assessment based on the Quality Model yield Meaningful Results?

Using the tool support, we conduct a quality assessment of open source systems. Then, we analyze the assessment results regarding *diversification* and *validity*.

Diversification describes whether the quality assessment yields different values for products with different quality levels. A commonly used measure for diversification in operations research is the normalized *entropy*, which origins from information theory: $E = -\frac{1}{ln(m)} \sum p_i * \ln(p_i)$, with $i = 1..m$, and p_i = probability to obtain value i.

That means, for calculating the entropy a scale of m values is needed. Therefore, the evaluation results $x \in \mathbb{D}$ will be mapped to the values 0..9. An entropy of 0 means there is no diversification at all. An entropy of 1 means that all assessment results are uniformly distributed. In between, at 0.8 lies the entropy of a normal distribution. Since it is a reasonable assumption that the "real" quality of all existing software products is normal distributed, the assessment result generated by our quality model should lie in the proximity of the normal distribution.

Validity describes that the assessment results are in concordance with the results obtained by another independent assessment approach as for example expert opinion. We used the expert-based quality assessment data of "Linzer Software-Verkostung" [8], which provides a ranking of five software products. Therefore the criterion *consistency* [1] can be applied. In our case, it will characterize the concordance between a product ranking based on the automatic assessment results using the quality model and the ranking independently provided by a group of experts. According to IEEE 1061 [1] we will use *Spearman's rank correlation coefficient* ρ to calculate the consistency between both rankings. Therefore we state the hypothesis H_0 which we want to show with $\alpha = 0.05$: H_0: $\rho(ranking_{QM}, ranking_{expert}) > 0$

5.2 Results

RQ 1: Suitability for Modeling Large Numbers of Static Code Checker Rules.

We created a quality model for software products written in the programming language Java, based on rules of the static code checkers *FindBugs* and *PMD*. The model contains 77 rules, 155 properties, 83 entities. There are 55 impacts describing the influence of properties characterizing product parts on the "-ilities" of the ISO 25010.

The hierarchy-based view of properties in the editor, enables efficient working and locating of previously modeled properties. This form of presentation is enabled by the well-defined relations between properties in the meta-model.

The support of the editor when editing models, as described in Section 4.1, proved valuable when working with it. Changes of a property does not render the model invalid, because the changes are propagated alongside the decomposition and refinement relations.

The form-based specification of the normalization proofed valuable as well. The editor determines all applicable ranges using the information of the quality model, and therefore prevents the user from modeling inapplicable ranges.

RQ 2: Meaningfulness of Quality Assessment Results.

The quality model was used to assess 17 open source systems. Besides analyzing the toolkit ConQAT itself, we selected the most downloaded *SourceForge* projects written in Java. Of these projects we included only those, where it was possible to compile the source code on our own with reasonable effort (less than one hour work per system). Table 1 shows the analysis results for a complete list of the systems. For each system its size in lines of code is listed,

[10]http://www.conqat.org/

to give an idea of their size; in sum about 8.5 MLOC. Furthermore, for each system the value of the root property [*Quality@Product*] is given.

Diversification. The mean of these values is $\mu = 0.813$ and the standard deviation $\sigma = 0.109$. The entropy for these results is $E = 0.637$, calculated like explained in Section 5.1.

Validity. Regarding the validity of the results, we compared the ranking of five open source systems produced by the quality model based assessment with an expert based ranking. The five systems used in the "Linzer Software-Verkostung" [8] and their rankings are shown in Table 2. We can see, that the rankings are almost consistent, only the systems *rssowl* and *log4j* are transposed. The *Spearman's rank correlation coefficient* for these two rankings is $\rho = 0.90$, which is close to a perfect correlation. Therefore, the hypothesis H_0 can be accepted on a satisfactory level of significance (p=0.045). This means, there is a significant positive correlation between the ranking provided by the base model and the experts.

5.3 Discussion

The case study shows that the meta-model and tooling allows to build a substantially large quality model, from static code analysis rules. The support of the editor proved valuable for effectively handling large models.

Regarding the results of the automatic quality assessment based on the quality model, we have shown that the diversification is satisfactory. Regarding the validity of the assessment results, we showed that there is a high correlation between the automatic determined ranking and an expert based ranking of the test systems.

5.4 Limitations

The generalizability of our results is limited because the number of assessed products (17 for diversification and 5 for validity) and their type (Java, Open Source) are limited. Regarding the internal validity, we cannot guarantee that the expert-based quality rating used as basis of comparison adequately represents the quality of the products.

Table 1: Quality Assessment Results

Nr	System	Lines of Code	Quality @Product
1	conqat-2.6	20,120	0.952
2	jabref-2.3	96,739	0.719
3	tomcat-6.0.24	321,904	0.670
4	tight-vnc-java-1.3.10	6,874	0.847
5	azureus-vuze-4.5.1.0	789,576	0.589
6	SweetHome3D-3.0	85,139	0.856
7	Freemind-0.8.1	78,308	0.772
8	liferay-portal-6.0.5	1,849,612	0.804
9	openproj-1.4	148,264	0.672
10	fckeditor-java-2.6	5,187	0.982
11	eclipse-SDK-3.7M4	3,806,681	0.734
12	apache-commons	512,383	0.828
13	hibernate-3.6.0	295,866	0.833
14	checkstyle-5.3	48,469	0.949
15	tv-browser-2.7.6	176,339	0.765
16	rssowl-2.0.6	133,799	0.913
17	log4j-1.2.16	43,018	0.898

Table 2: Ranking of Five Systems

System	QM Result	Ranking by QM	Ranking by Experts
checkstyle	0,949	1	1
rssowl	0,913	2	3
log4j	0,898	3	2
tv-browser	0,765	4	4
jabref	0,719	5	5

6. CONCLUSION AND FUTURE WORK

In this paper we presented a meta-model that defines an explicit structure for quality models. These quality models are suitable to model the high-level quality attributes and to relate them to metrics provided by existing static code analysis tools. Furthermore, the quality model allows to define aggregation specifications in order to enable automatic comprehensive quality assessments. Our tool-support allows for editing quality models and thereby supports the user through validity checks, wizards, and form-based templates for aggregation specifications. To obtain an overall quality assessment, our tooling can conduct automatic quality assessments based on the quality model. We presented a case study that shows that our model can be used to assess the quality of real software systems. Our results indicate that the model is capable of diversifying software systems of different quality levels. Moreover, we found that the assessment results are in line with the quality ranking conducted by independent experts. As future work we plan to extend our studies to software systems from other domains and programming languages. In addition, we intend to extend the quality model with more quality characteristics and measures.

7. ACKNOWLEDGMENTS

The presented work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "Quamoco, 01IS08023B".

8. REFERENCES

[1] IEEE Standard for a software quality metrics methodology. *IEEE Std 1061-1998.*

[2] ISO/IEC 25010 Systems and software engineering – System and software product Quality Requirements and Evaluation (SQuaRE).

[3] B. W. Boehm. *Characteristics of Software Quality.* North-Holland, 1978.

[4] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J.-F. Girard. An Activity-Based Quality Model for Maintainability. In *Proc. of the ICSM'07*, 2007.

[5] R. G. Dromey. A model for software product quality. *IEEE Trans. Software Eng.*, 21(2):146–162, 1995.

[6] B. Kitchenham, S. G. Linkman, A. Pasquini, and V. Nanni. The SQUID approach to defining a quality model. *Software Qual. J.*, 6(3):211–233, 1997.

[7] J. A. McCall, P. K. Richards, G. F. Walters, U. States, E. S. Division, A. Force, Rome Air Development Center, and S. Command. *Factors in Software Quality.* NTIS, 1977.

[8] R. Plösch. Software-Verkostung, `http://www.ipo.jku.at/dokumente/upload/Software\%20Verkostung\%20Impulsvortrag.pdf`, 23.04.2010.

[9] H. Schackmann, M. Jansen, and H. Lichter. Tool Support for User-Defined Quality Assessment Models. In *Proc. of MetriKon'09*. 2009.