This is the author's version of an IEEE-copyrighted article. The definitive version was published in Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE 2012), Dallas, USA (available at http://ieeexplore.ieee.org).

A Comprehensive Code-based Quality Model for Embedded Systems

Systematic Development and Validation by Industrial Projects

Alois Mayr, Reinhold Plösch
Department of Business Informatics
Johannes Kepler University
Linz, Austria
{alois.mayr1, reinhold.ploesch}@jku.at

Michael Kläs, Constanza Lampasona
Fraunhofer IESE
Kaiserslautern, Germany
{michael.klaes, constanza.lampasona}
@iese.fraunhofer.de

Matthias Saft
Corporate Technology
Siemens AG
Munich, Germany
matthias.saft@siemens.com

***Abstract*—Existing software quality models typically focus on common quality characteristics such as the ISO 25010 software quality characteristics. However, most of them provide insufficient operationalization for quality assessments of source code. Moreover, they usually focus on software in general or on information systems and do not sufficiently cover the particularities of embedded systems.**

We have developed a quality model that covers quality requirements for source code that are specific for embedded systems software. It provides comprehensive operationalization (with 336 measures) for C and C++ systems, which allows for largely automated quality assessments.

The empirical evaluations performed acknowledge moderate completeness of the requirements and the associated measures. Therefore, we still see room for improvements to allow covering even more aspects of embedded systems software quality. Nevertheless, the empirical validation (based on three industrial products) shows good concordance between the results gained by the automatic model-based assessment and independent expert judgment on code quality.

Keywords— quality assessment, embedded systems software, code quality, ESQM, SQUAD

I. INTRODUCTION

In recent decades, the integration of computer systems into various areas of daily life, e.g., medical instruments, power generation, or automotive components, has increased significantly. Potential malfunctions of *embedded systems* (ES) may harm human beings or the environment as described in [25]. More and more functionality of ES is provided by software, leading to a dramatic increase in software complexity. Therefore, software quality is of particular importance in designing and developing ES, even more so than in other application domains such as information systems [10].

In fact, various analytical or constructive approaches have emerged for assuring or improving quality during development, e.g., model-based development, dynamic testing, and static analysis techniques. They are seen as complementary rather than substitutive. In the model-based development of automotive ES, in particular, the generated code has lower defect density than manually implemented code and thus, the need for systematic code-centered automatic quality assessment is lower. Nevertheless, outside the automotive domain, especially for medical instruments, power generation, or industrial automation in general, embedded systems are typically coded manually, with only small fractions of the code generated from more abstract models. The VDC Research Group conducts extensive worldwide surveys of embedded development projects on a yearly basis and the 2010 survey [32] shows that C and C++ are still the leading languages for ES software development.

In this context, many companies are interested in applying *software quality models,* which can be a means for systematically analyzing and monitoring the quality of software and thus allow early feedback on quality [29][30].

Therefore, in the absence of an operationalized quality model addressing the particularities of ES, our objective is to develop an ES software quality model (ESQM) that provides the means for justified and comprehensive assessments of the source code quality of ES software in a quick and repeatable manner. The intended usage scenarios of ESQM are one-time assessments and comparison of different products as well as continuous controlling of the status of a product's internal software quality. Predicting quality is explicitly not an intended scenario. Additionally, we want to demonstrate the usefulness and practical applicability of ESQM by validating it in the context of industrial projects.

This work makes three major contributions. To the best of our knowledge, we provide (1) the first rigorously developed and validated quality model for ES software that is equipped with operationalization for C and C++ products; (2) a tested approach for iteratively developing a quality model for specific types of software, including an explicit meta-model that clearly structures the basic concepts of the quality model; and (3) support for performing largely automated quality assessments of ES source code, whose results appear to be consistent with the experience of a quality expert for the investigated industrial ES products.

In section II, we provide an overview of related work, with a focus on existing quality models for ES software. Next, in section III, we outline the four-phased approach used for developing the ESQM. In the four subsequent sections (IV to VII), each phase is described in more detail, including the objective of the phase, results, and performed validations. Finally, section VIII discusses threats to validity and section IX summarizes the presented work and provides directions for future research.

© 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This is the author's version of an IEEE-copyrighted article. The definitive version was published in Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE 2012), Dallas, USA (available at http://ieeexplore.ieee.org).

II. RELATED WORK

Software product quality has been a subject of research in various institutions and research groups pursuing similar goals. This has led to a number of *quality models* with diverse purposes and facets. Depending on the purpose, these contributions include models for the taxonomic specification of the term software quality (e.g., the models of McCall and Boehm discussed in [15], or the models in the standards ISO 9126 and 25010 [12]), measure-based quality models (e.g., the Maintainability Index [7]), prediction models (e.g., for quality [28], reliability [19] or defects [18]).

In [16], Kläs et al. present a comprehensive approach to classifying quality models and identify relevant ones in a goal-oriented way. The authors developed a systematic classification scheme for quality models and provided an exemplary landscape of existing quality models. Taking the proposed classification into account, we focus in this related work section on quality models that address the context of the *embedded systems domain*, consider the object *source code*, and support at least one of the following purposes: *specify*, *measure*, or *assess* software quality.

Although quality models that do not explicitly address the ES domain could theoretically also be used for assessing ES code, they are typically too abstract (e.g., [12]) or do not consider the programming languages most commonly used in this domain (C/C++) (e.g., [31]). Moreover, the specifics of ES software, e.g., limited memory consumption, are not sufficiently addressed by these models.

In the domain of ES software, there is some work on an *embedded software component quality model* (see e.g., [5], [2]). These contributions address the quality assessment of COTS components for ES software for certification and verification purposes. The authors extend and operationalize the ISO 25010 quality characteristics by defining measures with a focus on run-time or life-cycle characteristics. In consequence, the model neither covers internal code quality nor is it useful for assessing and controlling the quality of ES during development.

Wijnstra [33] identifies important quality attributes and aspects for medical products and stresses the need for multiple views on quality depending on different concerns. The article rather gives advice on architectural decisions in order to achieve certain quality attributes than providing a quality model that would be applicable in the development phase. Neumann et al. [24] propose a hierarchical quality model for ES that integrates the system view focusing on dependability with existing software quality models. Åkerholm et al. [1] present quality attributes that are important for ES software, in particular for automotive software, and emphasize how component technologies can contribute to achieving these attributes. Both Neumann et al. and Åkerholm et al. describe and elaborate interdependencies between identified quality attributes, but provide no measures for operationalizing them.

In contrast, *coding standards* for ES software (e.g., MISRA [21], [22], JSF AV [13]) focus on very detailed coding rules and thus provide clear specifications for measurements of ES source code. Moreover, various static code analysis tools (e.g., PC-lint or QA-MISRA) are available for checking the compliance of the source code with such coding rules. According to Ebert and Jones [10], static analysis provides a great value for defect prevention and removal in the source code of ES software. However, these coding standards are not covered by a quality model that would clarify the relevance of observed rule violations for specific quality characteristics or provide the option of tangible quality statements.

III. DEFINING THE EMBEDDED SYSTEMS SOFTWARE QUALITY APPROACH

Our main objective was to construct a quality model that facilitates both the description and assessment of ES software quality based on code properties. We agreed on a four-phase approach for developing the quality model, since this enables us to conduct early and explicit review and validation cycles (*quality gate* QG1 to 4). The purpose of the reviews is to check our intermediate results for comprehensibility and conformity, whereas validation checks for completeness and minimality of the model as well as appropriateness of the quality assessment results. Information gained by the review and validation at the end of each phase led to continuous improvement of the model. Furthermore, the reviews and validations helped us to gather and integrate the expertise of external experts (i.e., people from other business units or organizations with experience in embedded development and quality assurance) and allowed us to judge whether we were still on track. Since an exhaustive evaluation of all model elements would be too time-consuming for the external experts participating in the respective validation rounds, we focused in each round on a set of randomly chosen model excerpts. The exact number of excerpts considered in a round was determined based on the time expected to be needed for their evaluation.

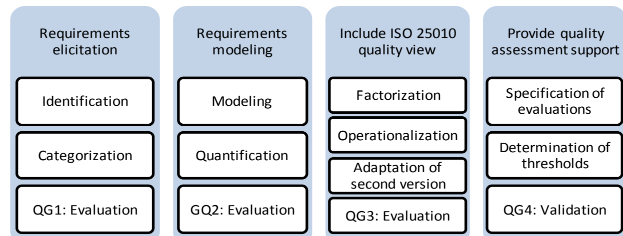

Figure 1: Approach for development of the quality model

As depicted in Figure 1, the emphasis of the first phase was on identifying and describing requirements for ES code quality. Afterwards, in phase two we modeled the requirements and associated proper measures. Providing measures is of vital importance as they are decisive for the applicability and usefulness of the model in practice. Next, in phase three we included the ISO 25010 quality characteristics and refined our model by introducing product factors. In the fourth phase, we focused on providing (semi-) automatic assessment support and thus added an assessment model, including proper evaluation and aggregation functions and rules. Moreover, we calibrated the assessment model by determining realistic thresholds for the assessment

© 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This is the author's version of an IEEE-copyrighted article. The definitive version was published in Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE 2012), Dallas, USA (available at http://ieeexplore.ieee.org).

functions. Altogether, three organizations worked on the quality model for three years. For modeling we used the Quamoco approach [31] and the corresponding meta-model, which were developed and evaluated simultaneously [17].

IV. Requirements for Embedded Systems Software Product Quality

The main objective of the first phase was to identify core quality requirements for ES software. Since the ISO 9126 as well as the ISO 25010 quality characteristics are very coarse-grained and abstract, it is difficult to directly derive requirements that are typical for ES code. Additionally, they do not provide assistance for completeness checks regarding (technical) requirements and measures once these are defined.

That is why we decided against refining the abstract ISO quality characteristics into measurable properties – the approach usually taken when developing quality models – but rather conducted a comprehensive literature analysis on commonly accepted guidelines and quality standards for embedded and safety-critical systems. These guidelines and standards are a good source for eliciting specific ES quality requirements as they directly provide (among other things) recommendations for ES code quality.

We examined the umbrella standard for functional safety, IEC 61508 Part 3 [11], as well as the standard for railway applications EN 50128 [6]. These standards play a vital role for safety-critical (embedded) systems and have led to generic and programming language independent requirements. The standards call for different kinds of requirements, such as organizational, process, methodological, or product requirements, with the latter being the most specific and code-related ones (e.g., IEC 61508-3 B.1.5 Limited use of pointers).

Since we decided to initially focus the operationalization of the quality model on C and C++ code, which is very common for ES software [10], we did not review standards or guidelines specific for other languages (e.g., Java). Indeed, we considered the MISRA-C [21], MISRA-C++ [22] guidelines as well as the JSF AV C++ coding standards [13] for programming language-specific requirements elicitation.

The majority of the rules defined in the mentioned guidelines address shortcomings of the C or C++ language, e.g., unspecified, undefined, or implementation-defined behavior. For instance, the MISRA-C rule 12.2 gives a fair warning about the unspecified order of evaluation of sub-expressions (e.g., `x=b[i]+i++;`). On the other hand, some rules target behavior that is well defined but may be dangerous in a safety context; for instance, the JSF AV rule 208 bans the usage of C++ exception handling.

Due to the different granularity and inconsistency of the requirements, we consolidated them. Finally, a total of 61 requirements remained from these sources.

> ***Exemplary requirement:*** *Avoid wrong and invalid references,* which is described as: *Wrong and invalid references are often introduced by initialization, pointer arithmetic, or wrong destructor implementation strategies. They have to be avoided to ensure that embedded systems run in defined (safe) states.* This requirement is linked to the MISRA-C rules 9.1, 11.1, 11.2, 17.6, 18.2, 18.3 and IEC 61508-3 B.1.5.

A. Embedded Quality Requirements

For reasons of clarity, we structured the 61 consolidated requirements along nine categories. Table 1 presents these categories and illustrates them with example requirements.

TABLE 1: Categories of Requirements

Notational requirements (NOT) address issues with respect to the style of the textual presentation of the source code, including naming and coding style conventions. *Example requirement:* NOT1 – Compliance with naming conventions, especially ensure unique naming
Declaration and definition requirements (DEC) address issues with respect to missing, unused, redundant, ambiguous, or error-tempting declared/defined symbols. *Example requirement:* DEC3 – Careful use of function-like macros
Procedural requirements (PROC) collect issues regarding (blocked) statements that are typically embraced by subroutines, e.g., control flow statements or expressions. *Example requirement:* PROC2 – Proper usage of switch statements
Memory requirements (MEM) bundle issues with statements related to memory references and pointers, e.g., memory allocation/de-allocation, overflow, or pointer arithmetic. *Example requirement:* MEM1 – Avoid wrong and invalid references
Protocol requirements (PROT) comprise requirements related to concepts that are implemented across statements, subroutines, or modules, e.g., error handling, concurrency, or permissions. *Example requirement:* PROT1 – Avoidance of exception handling
Design- and architectural requirements (DES) target requirements concerning the decomposition of a system into manageable and coherent sub-systems. Issues include modularity, encapsulation, and strong typing. *Example requirement:* DES4 – Usage of strong type systems
Correctness requirements (COR) cluster source code patterns that likely break the functional correctness of a system, e.g., unused/unfinished code blocks or unorthodox language usage. *Example requirement*: COR5 – Avoidance of unnecessary constructs
Timing requirements (TIM) collect implicit and explicit requirements for the time behavior of a system. *Example requirement:* TIM1 – Compliance with real-time requirements
Context-specific requirements (CON) contain topics that are dependent on the platform or application domain used. *Example:* CON1 – Proper usage of standard libraries and system platform/frameworks

B. Quality Gate 1: Review and Evaluation

In the first step of the first quality gate, all requirements were reviewed by several experts with respect to comprehensibility, appropriate level of abstraction, and consistent classification.

In the second step – after the integration of the review feedback – the completeness and appropriateness of the identified requirements, which become part of the model, were evaluated in a small empirical study conducted with five external experts.

We define *completeness* of the quality model with respect to requirements as the degree to which the quality model contains all mandatory ES quality requirements. We operationalize completeness as the ratio of the actual number of relevant requirements in the model and the total number of relevant requirements. The latter is refined in the sum of the actual number of relevant requirements in the model and the number of relevant requirements that are still missing. The number of missing requirements is then calculated based on inspection results of the external experts using a capture-recapture estimation model. Capture-recapture models are widely used in the field of biology to estimate population

© 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

size [4]. They were adopted in software engineering for inspections to predict the number of issues remaining after inspection in an artifact such as a requirements specification or source code documents [26]. We use it here to estimate the number of relevant but still not identified requirements.

Model *appropriateness* tells us whether and how the model covers specific mandatory elements and is evaluated subjectively by the external experts using a questionnaire.

Since 61 requirements are difficult to handle and check for completeness as a whole, we defined different perspectives (i.e., *quality goals*) in order to focus each evaluation task on a subset of requirements whose completeness is then evaluated with respect to the selected perspective. During the study, we proceeded as follows:

(1) First, we identified and internally consolidated a set of quality goals we consider as relevant for ES software (including predictability, limited resource usage, safety, certification, etc.), taking into account existing quality models – especially models for ES software – and the expertise available in the team. Then, based on internal discussion rounds, we associated each quality goal with all requirements whose fulfillment supports the specific goal (e.g., *MEM4 – Proper deallocation* supports the goals *predictability* and *limited resource usage*).

(2) Next, we conducted workshops where each external expert independently first created a list of quality goals (s)he considers as important for ES software and then compared this list with our quality goals in order to subjectively evaluate their appropriateness. In a final step, each expert reviewed the identified requirements for three (previously) randomly selected quality goals and noted requirements that (s)he missed or considered dispensable.

(3) Finally, the lists of missing requirements were used to estimate model completeness. In our capture-recapture based estimates, we used the Jackknife estimator [4], which assumes that missing elements may have different detection probabilities. In the context of software inspections, it provides the best estimation results compared to other estimators when applied with four or more reviewers [26].

<u>*Evaluation results*</u>*:* When we asked about important quality goals for ES software, *efficiency*, *reliability*, and *timeliness* were mentioned by three out of five participants. For all of them, the majority of the reviewers answered that they believed that the ESQM covers these characteristics appropriately. *Robustness* received two mentions as an important quality goal for ES software; *testability, persistency, reactivity*, and *tool chain issues* were each mentioned just once and were considered by the respective participant to be not appropriately covered by the model.

The sample of randomly selected quality goals contained *error handling*, *robustness*, and *code minimality*. As a result, we obtained a rate of completeness of 25% for the requirements with respect to the three considered goals. The reviewers' subjective estimation resulted in a rate of completeness of 29-62%.

<u>*Interpretation and improvement actions*</u>*:* The reasons for the low rate of completeness were found in the answered questionnaires. At the beginning, the requirements were formulated too strictly and tightly, with too much emphasis on source code. Following the experts' recommendations, we broadened the requirements to allow more coarse-grained requirements for which fully automatic measurement might be impossible.

V. Modeling Requirements and Measures

After eliciting and validating the requirements, the objective of the second phase was to make the requirements measureable and build an initial quality model.

Therefore, we associated the requirements with proper measures from static code analysis tools for C and C++. The majority of the measures are rules from the tool PC-lint, which provides preconfigured rule sets for quantifying MISRA rules. These rule sets served as a basis for the identification of proper measures for the requirements. As a result, at this time the quality model had to provide means for measures and requirements.

A. Meta-Model

Based on the needed concepts identified above, the resulting model structure was pretty simple. It consists of a requirements hierarchy and measures. A measure may be suitable for quantifying more than one requirement and vice versa. The meta-model allows modeling hierarchies of requirements, such as the structure illustrated in Figure 2.

B. Embedded Quality Model – First Version

The first version of the quality model consisted of 61 requirements. These requirements were structured along nine categories. A total of 304 measures were modeled and assigned to the requirements (multiple times). Figure 2 illustrates the structure on a small excerpt of the model.

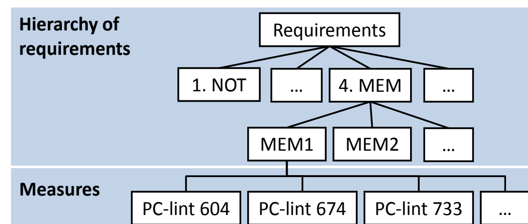

Figure 2: Requirements hierarchy with assigned measures (excerpt)

One example is *MEM1 – Avoid wrong and invalid references* which is quantified by 19 PC-lint rules. Moreover, one out of four requirements is quantified with more than 10 measures. The high number of measures for this requirement is hard to handle cognitively. In Moody's work on decomposition principles for (data) models, the author calls (among others) for cognitively manageable pieces of decompositions [23]. According to Miller's "seven plus or minus two" rule, the human mind can only handle a maximum of nine concepts at a time without exceeding the limits on short-term memory [20].

On the other hand, almost a quarter of the requirements we identified had no associated measures at this time. The reason for this is that we defined some requirements that are important for ES software but difficult or impossible to quantify using static code analysis tools. In the third phase, we addressed these problems.

© 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

C. *Tool Support for Quality Specification*

Quality modeling is a time-consuming activity, as it requires thorough understanding of the abstract concept of quality regarding a specific context or view as well as all facets and difficulties related to its decomposition and measurement. Therefore, proper tool support is essential; it enables the modelers to develop quality models step by step and collaboratively. Furthermore, the tool should allow turning on/off any consistency constraints defined in the meta-model (e.g., the number of measures associated with a requirement) to give the modelers flexibility during the modeling progress.

Therefore, we used the Quamoco quality model editor as presented in [8]. The editor is based on the Eclipse Modeling Framework, which allows quick and easy reaction to any changes of the meta-model.

D. *Quality Gate 2: Review and Evaluation*

In the first step of the second quality gate, the model itself was reviewed with respect to consistency with the defined modeling guidelines and the measures with respect to comprehensibility.

In the second step – after the review feedback was integrated – the completeness of the identified measures with respect to requirements was evaluated in a small study conducted with five external experts. Since it would be too time consuming to evaluate the completeness of the measures for all requirements, we randomly selected a sample of five requirements, which were evaluated to get an impression of the overall completeness.

In order to do this, we conducted workshops during which each external expert independently identified for each of the five randomly selected requirements all measures that (s)he missed or considered dispensable using a questionnaire.

After analyzing the results of the questionnaires, we estimated the completeness of the measures for each of the selected requirements by again applying a capture-recapture model with a Jackknife estimator (cf. Section 0). Note that instead of evaluating whether the model contains all relevant requirements needed to address relevant ES software quality goals (see quality gate 1), this time we evaluated whether all aspects of the requirements were sufficiently covered by associated measures.

Evaluation results: The randomly selected requirements fell into the categories *PROT, DES, MEM, PROC,* and *DEC*. As a result, we obtained a rate of completeness of 64% for the measures with respect to the given sample of requirements. The reviewers' subjective estimations of completeness were between 77% and 100%.

The measures *code comment ratios* and *avoiding pointer arithmetic* were considered to be unsuitable for the ESQM, i.e., they were considered of no relevance for ES software.

Interpretation and improvement actions: Similarly to the first phase, we analyzed the experts' comments and enhanced the respective requirements. Additionally, we introduced further measures, mostly manual ones, in order to cover requirements that had not been quantified yet or to improve requirements that were not covered sufficiently.

VI. ADDING FACTORS AND ISO 25010

The objectives of the third phase were to make the model more programming language independent by abstracting from concrete measures and adding the quality characteristics of ISO 25010 [12] to the model as a common view on quality.

We deliberately disregarded the ISO 25010 for requirements elicitation during the first phase since the provided model is too abstract and does not focus on ES software. However, at this time, we were already aware of the specific quality requirements and associated measures for ES software. We integrated this popular view on software quality because higher-level management is typically less interested in how well quality requirements are addressed, but rather in whether the software might have problems regarding certain quality characteristics as defined in ISO 25010 (reliability, security, etc.).

We observed that requirements are a good means for deriving relevant measures for operationalization, but they are too coarse-grained to link all measures pooled by a requirement to a specific set of quality characteristics (remember also that one fourth of the requirements is quantified by more than 10 measures). On the other hand, the measures are typically too detailed and programming language specific to allow direct assignment to the quality characteristics. Furthermore, this would have made the quality model very complex, since a lot of measures were associated with each quality characteristic. Additionally, the reusability and maintainability of the quality model itself would have suffered because each newly added measure had to be associated with one or more quality characteristics and one or more requirements.

As a consequence, we needed an intermediate level of abstraction in order to ensure programming language independence and keep the model cognitively manageable. Therefore, we introduced the basic concept of a *product factor*, which describes a property of an entity. This concept is similar to Dromey's [9] quality carrying properties of product components and also used by the Quamoco quality model in a slightly modified form [31]. An entity expresses the part of a product (source code) that should have a particular property.

Exemplary product factor: *Reference Validity @Assignment Statement*, where Assignment Statement is the entity and Reference Validity is the property. This product factor means that the memory address (i.e., the reference) assigned to a pointer variable by an assignment statement should always match a safe location in the memory (i.e., be valid).

This seems very detailed and technical but provides a clear understanding of what properties are demanded from a specific source code entity and helps to structure and abstract from even more detailed measures. In order to use these product factors as a basis for specific views on quality, in our case the ISO 25010 quality characteristics tree, the quality model additionally needed means for defining justified relations between product factors and quality aspects. We use the more general term *aspect* to clarify that not only quality characteristics of the ISO 25010 quality model can be

© 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This is the author's version of an IEEE-copyrighted article. The definitive version was published in Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE 2012), Dallas, USA (available at http://ieeexplore.ieee.org).

modeled here, but also arbitrary views on quality. This is done by means of a meta-model element called *impact*.

A. Enhancements of the Meta-Model

The enhancements to the quality model sketched above required significant refinements to be made to the meta-model.

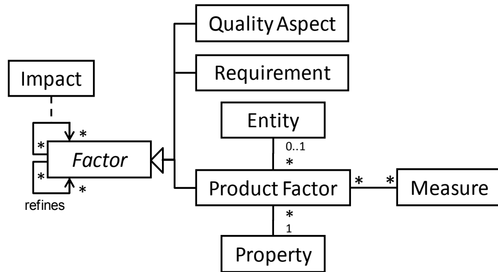

Figure 3: Enhanced meta-model

The central element of the revised meta-model is the (abstract) *factor*, which can be derived to allow more specific factor types. Each specialized type relates to its own hierarchical view on quality. The ESQM provides a hierarchy of quality aspects (i.e., ISO characteristics), requirements, and product factors. Factors can refine only other factors of the same type, e.g., quality aspects can only refine other quality aspects. The *impact* relation is directed and states whether a factor positively or negatively influences another one and provides a justification for this kind of impact. Impact relations are only allowed between factors of different types, e.g., product factors can only have an impact on requirements or quality aspects. For this purpose, the meta-model considers various constraints that ensure such modeling restrictions.

Product factors play a special role in the meta-model. A product factor can refer to an *entity* and belongs to a *property* that the product factor describes. They are the only factors that are associated with measures and are seen as the foundation of each quality model – a detailed but programming language independent abstraction layer. Like any other factors, the product factors can be composed into more abstract ones using the refine relation (modeled at the factor). The product factors are intended to serve as the source for all impacts on other factor hierarchies.

B. Enhancements of the Model

Figure 4 provides an overview of an excerpt of the ESQM. The measures quantify the product factors, which again refine other product factors or impact the requirements or quality aspects.

After reworking the requirements of the first version of the quality model, it now consisted of 60 requirements. We added further measures, mostly manual ones (22), in order to close the gap of requirements not yet quantified.

Basically, the ISO standard refines its top-level quality characteristics into sub-characteristics. Unfortunately, the sub-characteristics sometimes overlap or are interdependent, as the standard itself notes, e.g. "Availability is therefore a combination of maturity (which governs the frequency of failure), fault tolerance and recoverability (which governs the length of down time following each failure)" [12]. All of these are sub-characteristics of reliability. In such cases, the product factors impact top-level ISO characteristics instead of its sub-characteristics. However, the refinements of maintainability, functional suitability, and performance efficiency are better suited for impacts of product factors and therefore impacted at the level of sub-characteristics.

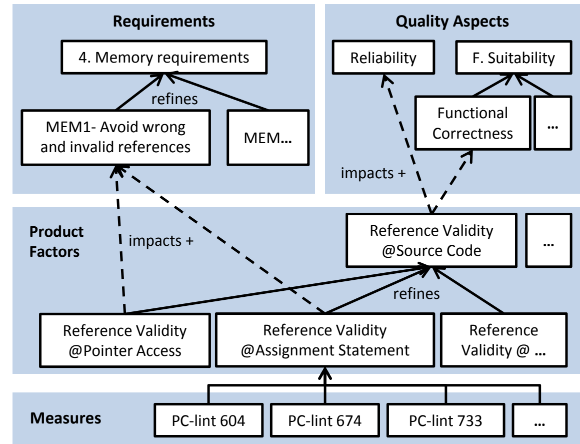

Figure 4: ESQM quality model with factors (excerpt)

In total, the model provides factors of 32 different properties, such as *Behavioral Integrity, Encapsulation Strength, Reference Validity, Unintentional Side-Effect,* or *Uselessness*. These properties can be combined with 87 entities, such as *Class, Field, Subroutine Parameter, Assignment Statement,* or *Switch Statement*. On this basis, we modeled 162 product factors on the leaf level of the product factor hierarchy.

This additional level of abstraction led to fewer measures pooled by a product factor. In contrast to the first version (one out of four requirements), only one product factor now had more than 10 measures assigned to it. On the other hand, one half of the product factors at the leaf level were quantified by exactly one measure. In our view this is not a problem, since adding further measurement tools or measures for other languages in the future will increase the number of measures for these factors.

C. Quality Gate 3: Review and Evaluation

In the first step of the third quality gate, the model itself was reviewed with respect to consistency with the defined modeling guidelines and comprehensibility of the added elements (i.e., in particular, the quality aspects, product factors, and impact relationships).

In the second step – after the review feedback was integrated – the completeness of the identified product factors and the quantifying measures was evaluated in a final study conducted with six external experts. Since we could not evaluate the completeness of all 336 measures with respect to the quality requirements, we randomly selected a sample of five quality requirements and evaluated the completeness of the measures that were indirectly associated via product factors with these requirements. Additionally, we evaluated the completeness of the introduced product factors with respect to three randomly selected quality aspects.

© 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This is the author's version of an IEEE-copyrighted article. The definitive version was published in Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE 2012), Dallas, USA (available at http://ieeexplore.ieee.org).

Evaluation results: The randomly selected requirements fell into the categories *CON, DEC, DES, PROC,* and *PROT*. The selected requirements were different from those selected in quality gate 2. As a result, we obtained a rate of completeness of 54% for the measures with respect to the requirements. The reviewers' subjective estimations of completeness were between 68-98%.

Additionally, we analyzed the completeness of the product factors with respect to three ISO quality aspects: *reliability, reusability,* and *time behavior.* We obtained a rate of completeness of 47%. The reviewers' subjective estimations of completeness were between 51% and 94%.

Interpretation and improvement actions: To our surprise, the completeness of the measures regarding the inspected requirements was lower (54%) than the one of the previous version of the model (64%). One possible explanation may be the changed set of inspected requirements. Another explanation may be that the additional abstraction layer introduced with the product factors helped the experts to identify missing measures more effectively. Furthermore, the focus of ESQM in the current version is on static analysis, but the experts often demanded other aspects of quality, such as dynamic analysis and tests.

A comparison of the completeness of the requirements regarding the quality goals (25%) in the previous version with the completeness of the product factors regarding the quality aspects (47%) yields a higher degree of completeness on the intermediate layer of the model. Once again, we analyzed the experts' comments and enhanced the respective product factors and introduced additional measures.

VII. Adding Assessment Support

After the completion of the third phase, the quality model provided a specification and low-level quantification of quality for ES software. In the fourth phase, we addressed the operationalization of the quality model regarding (semi-) automatic quality assessments to get interpretable quality statements on higher abstraction levels. Therefore, we developed and integrated an assessment model that enables collecting required measurement data, evaluating the collected data, and aggregates these results according to the developed quality model. Additionally, we adapted the open-source quality analysis toolkit ConQAT [8] to cope with the requirements that emerged from our assessment model.

A. Enhancement of the Meta-Model

In order to support the planned kind of quality assessments, the meta-model had to be extended by two additional elements, instrument and evaluation.

Instruments are associated with measures and decide how the measures defined in the quality model are actually collected. Some measures may be collected based on a described manual procedure because they cannot be collected automatically or tooling is not available; others are collected using the output of a specific source code analyzer.

Evaluations are complex model elements that are associated with a factor for which they provide an evaluation result. In order to do this, they can use (a) the results of associated measures if the factor is a product factor or (b) the evaluation results of subordinate factors (i.e., factors refining or impacting the evaluated factor). In case (a), they are called *measure evaluations* and are responsible for normalizing the measurement results, mapping them onto a common evaluation scale and aggregating the obtained evaluation results to provide an evaluation result for the evaluated factor. In case (b), they are called *factor aggregations* and are responsible for aggregating the evaluation results of all factors refining or impacting the evaluated factor.

B. Assessment Method

In order to support theoretically sound and comprehensive assessment of software quality using hierarchical quality models, Trendowicz et al. [27] developed a quality assessment method called SQUAD. The method was developed to addresses the 15 most relevant requirements on software quality assessments as identified by a survey and literature review. Based on a review of existing quality assessment methods, the SQUAD method proposes a combination of established concepts from the field of Multi Criteria Decision Analysis [14]. It defines the semantics of the evaluation elements in a quality model and provides guidelines for the concrete operationalization of a model for quality assessments. Based on the positive experience we gained when the method was applied to operationalize the general Quamoco quality model [31], we used the method for our quality model, too.

C. Operationalization of the Model

In the following, we briefly describe the key components of the quality assessment based on SQUAD and how we operationalized our quality model for quality assessments.

Normalizing Measurement Results: When assessing the quality of a software product, the first step is the collection of the required measurement data as specified by the measures in the quality model. Such a measure might be the number (and location) of undocumented functions in the source code. However, this information in isolation can usually not be evaluated reasonably. For instance, 20 undocumented functions might be okay if the software has several thousand functions, but might not be acceptable for a software product with only 50 functions or less. Moreover, the size of the affected functions also plays a role; if the undocumented functions are small helper functions with five or fewer lines of code, the evaluation would be expected to be better than if each of the affected functions consists of 100 or more lines of code.

This means that in order to make the measured values comparable among different software products, the measurement results have to be normalized. For measures providing a list of findings (i.e., problematic places in the source code), we calculate, for instance, the relative code proportion affected. For the measure 'undocumented functions', we would calculate the ratio between the amount of source code of the undocumented functions and that of all functions in the product. Typically, one person decides how a given measure is normalized. We made this more rigorous by using guidelines and a four-eye principle to consistently apply normalization guidelines for all 336 measures.

© 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This is the author's version of an IEEE-copyrighted article. The definitive version was published in Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE 2012), Dallas, USA (available at http://ieeexplore.ieee.org).

Evaluating Normalized Results: The resulting normalized value for a measure is comparable among different software products but can usually not be compared with the values of other measures, e.g., how to compare the result that 40% of the functions are insufficiently documented with 1% of the code containing functions with unused parameters? Thus, each measure has to be evaluated independently and thereby translated into a value on a common evaluation scale in order to allow us to aggregate the results of different measures.

The normalized measurement result of a measure is evaluated by applying an *evaluation function*, which maps the normalized measurement results onto the common evaluation scale [0,1]. In order to keep the assessment easy to understand and interpret, we limited the applied evaluation functions to *linear increasing* and *linear decreasing* functions with a min and max threshold. The higher the normalized measurement result, the lower the evaluation result would be if the linear decreasing function is used.

Besides the decision about whether to use a decreasing or increasing evaluation function, which was made by two model developers together based on the relationship between the measure and the evaluated factor (e.g., a high value for 'undocumented functions' decreases the evaluation results of the factor 'Moderate Occurrence @Subroutine Comment'), the min and max values have to be defined for each function. Since determining these thresholds for all measures based on expert judgment would be time consuming and unreliable, we decided to use a benchmarking-based approach: For each measure, we collected normalized measurement data from 17 open source products (8 written in C & 9 written in C++), identified potential outliers in the data using box plots as a robust approach, and taken the minimum and maximum of the non-outlier values as the min and max thresholds for the respective evaluation function (Figure 5).

This approach also implicitly addresses the problem of falsely reported findings of static code analyzers (i.e., false positives). They are automatically reflected in the min/max thresholds since rules detecting many false positives receive higher min/max values than reliable rules with fewer ones.

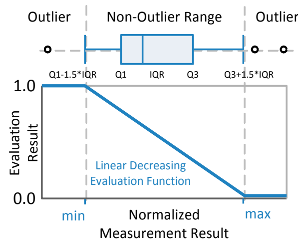

Figure 5: Calibration of evaluation functions
(Qx: Quartile x, IRQ: Inter-Quartile Range)

In the cases where an insufficient number of non-zero values were available for reliably determining a min and max value (i.e., the evaluated measure detected issues in fewer than four products), we set min = 0 and max = 10^{-7}, which approximates a jump function.

Evaluating Factors: In order to evaluate a factor, we have to consider the evaluation results of its subordinated factors or measures as well as their relative importance. In SQUAD, a weighted sum approach is used to aggregate the evaluation results of the subordinates and calculate the evaluation result for the evaluated factor.

This means that– except for the trivial case of factors quantified by one measure – we had to determine weights for the subordinated measures or factors based on their relative importance. We did this by providing for each product factor and quality aspect a ranking of its subordinated measures or factors. The ranking was done by two model developers with experience in the ES domain and in the measures used in the model. They classified all subordinate factors or measures into three priority classes at maximum. The subordinate factors or measures pooled in the priority classes were ranked with the respective priority. Then we used the Rank-Order Centroid method [3] to transform the ranking results into a set of weights $w_{i=1\ldots n} \in [0,1]$ that sums up to 1. In consequence, the calculated sum and thus the evaluation result is again a value on the evaluation scale [0,1].

Interpreting Evaluation Results: Because a value on the [0,1] evaluation scale is difficult to interpret for an human assessor, one can apply an *interpretation function*, which maps the result on the evaluation scale to a more intuitive interpretation scale. In our case, we used German school grades, where 1 is very good and 6 unsatisfactory. According to Figure 6, if the evaluation result is lower than 0.29, the product gets the worst grade. The best grade is given if the evaluation result exceeds 0.91. In order to allow assessors to observe minor quality changes, we interpolate the grades for higher precision, i.e., we provide grades with one decimal place (such as 3.8 or 5.2).

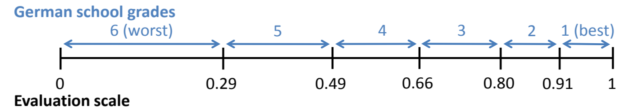

Figure 6: Mapping of evaluation values to school grades

D. *Quality Gate 4: Review and Validation*

The focus of this validation was to check whether the automatic assessment results provided by the ESQM are in concordance with results obtained by another independent and valid approach for assessing product quality.

For this purpose we compared the results of the automatic assessment for three industrial products with the independent judgment of one professional who knows these three products well from a quality perspective. His judgments are based on detailed code reviews he performed for each of the products that took several days.

Validation results: The professional rated the product quality with school grades using a given set of quality criteria similar, but not totally conformant to the quality characteristics of ISO 25010. Therefore, not all assessment results on the quality aspect level can be compared directly; however, they are provided for the sake of completeness (gray areas in Table 2). The overall quality judgment of the professional was calculated from the median of his judgment for the quality attributes maintainability, predictability, resource utilization, safety, and security. It shows a clear order of the quality of the products, with product A being the best, product C being second, and product B being the worst.

© 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

TABLE 2: COMPARISON OF EXPERT JUDEMENT AND THE ESQM ASSESSMENT RESULTS (GERMAN SCHOOL GRADES)

	Expert Judgment					
Product	***Quality***	*Maintain ability*	*Predict ability*	*Security*	*Safety*	*Resorc es Util.*
A	**1**	1	2	1	1	3
B	**5**	5	5	5	5	4
C	**3**	3	3	3	3	1

	ESQM Assessment Results Using SQUAD					
Product	***Quality***	*Maintain ability*	*Reli ability*	*Security*	*Functio nality*	*Perfor mance*
A	**1.2**	1.1	1.6	1	1.1	1
B	**3.1**	2.9	3.5	2.4	3.4	2.3
C	**1.8**	1.4	2.3	1.4	1.4	1.4

Interpretation: The ESQM-based assessment results for the three products are similar to the expert's judgment, i.e., the order of products is the same. However, the ESQM-based assessment calculates better grades, although it keeps the same order as the expert. One reason for this might be that the ESQM quality model focuses on quality requirements that are directly related to the particularities of ES and the model does not consider more general quality factors that are independent of a specific application domain (e.g., adherence to naming conventions). We assume that the expert considered these more general quality aspects in his rating.

The sub-characteristics used by the professional are slightly different from the ISO 25010 quality aspects used in the automatic assessment. Nevertheless, some detail results can be compared. For maintainability (which is directly comparable), ESQM arrives at the same quality order of the products as the expert. Reliability of the ESQM model can be related to the expert opinion on predictability and the ESQM assessment arrives at the same quality order of the products as the expert. The quality aspect security is directly comparable and again ESQM yields the same order for the quality of the products as the expert.

VIII. THREATS TO VALIDITY

Our main validation results for the ESQM are based on two different types of evaluations: the completeness evaluations conducted in QG1 to QG3 for different model elements and the criteria validity evaluation conducted in QG4. Next, we discuss the threats to validity that we consider to be most relevant for each of the two types of evaluations.

Completeness evaluations: A major threat to validity is the fact that due to the low number of external experts (5-6) in each inspection round, statistical tests with a reasonable power level could not be performed to support the validity of our completeness estimates (*conclusion validity*). Moreover, the participants were a convenience sample because they were external to the project and could not be forced to participate (*external validity*). Finally, the samples of the inspected model elements (although randomly selected) might not be representative of the complete model due to the low number of elements (*internal validity*). Although these facts have an impact on the validity of our results, they could not be avoided due to the given design constraints: finding a higher number of appropriate (experienced) external experts who would agree to spending more than half a day on inspecting a sample of quality model elements could not be realized in our context.

Criterion validity evaluation: There are four major threats to our criterion validity study. First, due to the low number of assessed products, we could not statistically test the agreement between the given expert judgment and the ESQM results (*conclusion validity*). Second, only one expert rated the three products; therefore, we cannot be sure that the criterion to which we compare our ESQM results is a valid representation of our construct 'code-based ES quality'. Unfortunately, it is very difficult to obtain quality statements for the same industrial embedded products from multiple professionals. The professional who did the rating needed several days for each product, and software failure or bug numbers usually do not represent a valid measure for code-based software quality. Third, since the expert's judgment on the three products was done completely independent from the model development, the quality aspects considered do not match perfectly between the expert-based and the ESQM-based assessments (*construct validity*). Fourth and finally, all assessed products were from only one company; hence, it is not clear to which extent our results can be generalized to ES products in general (*external validity*).

IX. SUMMARY AND FUTURE WORK

Source code quality of ES software is of vital importance as software-equipped ES have become more and more established in daily life in recent decades. Unfortunately, in practice there is a lack of operationalized quality models that cover the specifics of ES code quality. Therefore, our work provides three major contributions. We (1) systematically developed and validated a quality model that covers the specifics of ES code quality and provides operationalization for products written in C and C++, (2) we present a tested approach for developing a quality model for a specific domain (i.e., ES software), and (3) provide support for largely automated quality assessments.

The developed quality model (ESQM) covers a set of identified quality requirements that are specific for ES software. Additionally, it provides programming language independent factors for bundling similar measures of various tools and abstract from specific languages. In our evaluations we focused primarily on (a) completeness of the specified quality model, which was approximated considering the results of the inspection-based capture-recapture estimates: completeness of measures with respect to the requirements (54%) and completeness of the product factors with respect to the ISO quality characteristics (47%). The numbers may seem moderate with 100% in mind, but we do not know of any other quality model development where the model completeness was quantified and as rigorously checked by external experts as in our case. Therefore, our completeness numbers should rather be considered as a baseline against which future modeling efforts can be compared. In order to determine the usefulness of the quality model in practice, we performed an initial evaluation of (b) the validity for the ESQM-based assessment results. Although being more repeatable and requiring considerably less effort than manual code inspections, we observed concordance between the

© 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This is the author's version of an IEEE-copyrighted article. The definitive version was published in Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE 2012), Dallas, USA (available at http://ieeexplore.ieee.org).

ESQM-based ranking and independent expert judgment on the quality of three assessed industrial products. This means that the products identified by the expert as being the best, middle, and worst one based on manual code inspections are the same as those calculated by the automatic approach.

Due to the low number of trial products, we cannot test these results for statistical significance, but they provide a first promising clue for good criterion validity of the ESQM-based assessments. Based on these results, the model is currently being applied in a company to get a quick picture of the quality of ES software and motivate and focus code-related improvement actions.

In the future, we plan to improve the existing model through better automation of some manual measures with additional tools. Furthermore, it seems promising to involve dynamic analysis techniques and tools to determine test coverage or performance issues.

ACKNOWLEDGMENT

This work was funded in part by the German Federal Ministry of Education and Research (BMBF) in the context of the grant "Quamoco, 01IS08023B/C".

REFERENCES

[1] M. Åkerholm, J. Fredriksson, K. Sandström, and I. Crnkovic, "Quality Attribute Support in a Component Technology for Vehicular Software," in *Fourth Conference on Software Engineering Research and Practice in Sweden Linköping Sweden*, pp. 1-9, 2004.

[2] A. Alvaro, E. Almeida, and S. Meira, "Quality attributes for a component quality model," in *10th WCOP/19th ECCOP*, 2005.

[3] F. H. Barron and B. E. Barrett, "Decision quality using ranked attribute weights," *Management Science*, vol. 42, no. 11, pp. 1515-1523, 1996.

[4] K. P. Burnham and W. S. Overton, "Estimation of the Size of a Closed Population when Capture Probabilities vary Among Animals," *Biometrika*, vol. 65, no. 3, pp. 625–633, Dec. 1978.

[5] F. Carvalho and S. Meira, "Towards an Embedded Software Component Quality Verification Framework," *2009 14th Int. Conf. on Engineering of Complex Computer Systems*, pp. 248-257, 2009.

[6] CENELEC, "EN 50128: Railway applications - Communications, signaling and processing systems - Software for System Safety." 2001.

[7] D. Coleman, D. Ash, and B. Lowther, "Using metrics to evaluate software system maintainability," *COMPUTER,*, vol. 27, no. 8, pp. 44-49, 1994.

[8] F. Deissenboeck, L. Heinemann, M. Herrmannsdoerfer, K. Lochmann, and S. Wagner, "The quamoco tool chain for quality modeling and assessment," in *33rd International Conference on Software Engineering (ICSE)*, pp. 1007–1009, 2011.

[9] R. G. Dromey, "A model for software product quality," *IEEE Trans. on Software Engineering*, vol. 21, no. 2, pp. 146-162, 1995.

[10] C. Ebert and C. Jones, "Embedded Software: Facts, Figures, and Future," *Computer*, vol. 42, no. 4, pp. 42-52, 2009.

[11] International Electrotechnical Commission, "IEC 61508: Functional safety of electrical/electronical/programmable electronic safety-related systems," 2010.

[12] International Organization for Standardization, "Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE)," 2005.

[13] JSF, "Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program," Lockheed Martin Corporation, 2005.

[14] R. L. Keeney and H. Raiffa, *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Cambridge University Press, 1993, p. 592.

[15] B. Kitchenham and S. L. Pfleeger, "Software Quality: The Elusive Target," *IEEE Software*, vol. 13, no. 1, pp. 12-21, 1996.

[16] M. Kläs, J. Heidrich, J. Münch, and A. Trendowicz, "CQML Scheme: A Classification Scheme for Comprehensive Quality Model Landscapes," *35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 243-250, 2009.

[17] M. Kläs, C. Lampasona, S. Nunnenmacher, S. Wagner, M. Herrmannsdörfer, and K. Lochmann, "How to Evaluate Meta-Models for Software Quality?" *Joined International Conferences on Software Measurement. IWSM/MetriKon/Mensura*, Shaker, pp. 443-462, 2010.

[18] M. Kläs, H. Nakao, F. Elberzhager, and J. Münch, "Support planning and controlling of early quality assurance by combining expert judgment and defect data—a case study," *Empirical Software Engineering*, vol. 15, no. 4, pp. 423-454, 2010.

[19] Lyu, M. R., "Software Reliability Theory" in *Encyclopedia of Software Engineering*. John Wiley & Sons, 2002.

[20] G. A. Miller, "The magical number seven, plus or minus two: some limits on our capacity for processing information.," *Psychological review*, vol. 63, no. 2, pp. 81-97, 1956.

[21] MISRA, "MISRA-C 2004 Guidelines for the use of the C language in critical systems," Motor Industry Research Association, 2004.

[22] MISRA, MISRA C++ 2008 Guidelines for the use of the C++ language in critical systems, Motor Industry Research Association, 2008.

[23] D. Moody, "A decomposition method for entity relationship models: a systems theoretic approach," *International Conference on Systems Thinking in Management*, pp. 462-469, 2000.

[24] R. Neumann, L. Grunske, and B. Kaiser, "Hierarchical Software Quality Models - A step towards quantifying non-functional properties," in *Proceedings of the 12th International Workshop on Software Measurement*, pp. 107-124, 2002.

[25] C. Perrow, *Normal Accidents: Living with High Risk Technologies*, 2nd ed., Princeton University Press, 1999.

[26] H. Petersson, T. Thelin, P. Runeson, and C. Wohlin, "Capture-recapture in Software Inspections after 10 Years Research – Theory, Evaluation and Application," *Journal of Software and Systems*, vol. 72, no. 2, pp. 249-264, 2004.

[27] A. Trendowicz, M. Kläs, C. Lampasona, J. Münch, C. Körner, and M. Saft, "Model-based Product Quality Evaluation with Multi-Criteria Decision Analysis," *Proceedings of the Joined Int. Conf. on Software Measurement (IWSM/MetriKon/Mensura)*, pp. 3-20, 2010.

[28] S. Wagner, "A Bayesian network approach to assess and predict software quality using activity-based quality models," in *Information and Software Technology*, vol. 52, no. 11, pp. 1230-1241, 2010.

[29] S. Wagner, K. Lochmann, S. Winter, A. Goeb, and M. Klaes, "Quality Models in Practice. A Preliminary Analysis," *3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE Computer Society, 2009.

[30] S. Wagner, K. Lochmann, S. Winter, A. Goeb, M. Kläs, and S. Nunnenmacher, "Software quality in practice - survey results," 2010 [Online]. Available: https://quamoco.in.tum.de/wordpress/wp-content /uploads/2010/01/Software Quality Models in Practice.pdf

[31] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Ploesch, A. Seidl, A. Goeb, J. Streit, "The Quamoco Product Quality Modelling and Assessment Approach," in *Proceedings of the 34th Int. Conf. on Software Engineering (ICSE)*, pp. 1133-1142, 2012.

[32] "What languages do you use to develop software? - On Target: Embedded Systems," 2010. [Online]. Available: http://blog.vdc research.com/embedded_sw/2010/09/what-languages-do-you-use-to-develop-software.html. [Accessed: 01-Feb-2012].

[33] J. G. Wijnstra, "Quality attributes and aspects of a medical product family," *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, vol. 0, no. c, p. 10, 2001.

© 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.