Quality Plans for Measuring Testability of Models

Hendrik Voigt [1], Baris Güldali [2], Gregor Engels [1,2]

[1]Department of Computer Science, [2]Software Quality Lab / University of Paderborn / Germany

Abstract

For models used in model-based testing, the evaluation of their testability is an important issue. Existing approaches lack some relevant aspects for a systematic and comprehensive evaluation. Either they do (1) not consider the context of software models, (2) not offer a systematic process for selecting and developing right measurements, (3) not define a consistent and common quality understanding, or (4) not distinct between objective and subjective measurements.

We present a novel quality management approach for the evaluation of software models in general that considers all these aspects in an integrated way. Our approach is based on a combination of the Goal Question Metric (GQM) and quality models. We demonstrate our approach by systematically developing a short quality plan for measuring the testability of software models.

1 Introduction

Model-based testing (MBT) advocates the systematic use of software models in particular for testing activities, e.g. generation of test cases, test oracles, and test execution environments [HL03]. If a software model is well suited for testing activities, we speak of a *testable* model. Thus, *testability* of software models is an important quality indicating the degree to which a software model facilitates testing activities in a given test context [Jun99]. The testability of software models should already be analyzed and improved at modeling time before MBT activities begin. This helps to save costs for the detection and correction of testability defects in later phases.

UML is the de facto modeling language in software development. [Bin99] explains how different UML diagrams can be used in testing activities. Different approaches for evaluating testability of UML models have been suggested by [Bin99, MS01, BTS02, BL01]. Most of the approaches specify testability requirements using checklists or set of metrics or some informal specifications. For instance, Binder defines in [Bin99] some structural, syntactical and semantical requirements on UML statecharts if they are used for test case generation. He captures these requirements in five checklists. Using these checklists, a modeler can be aware of certain properties of statecharts if he aims to use them for testing.

The existing approaches evaluating testability of software models mainly lack the following relevant aspects [VE08]:

1. *Context characterization is missing:* The testability of software models strongly depends on their context, in which they are analyzed and interpreted. For example, if only statecharts are available it is impossible to measure removed states by a subclass. Hence, the context of a software model influences the measurement of its testability.
2. *Systematic process is missing:* Existing evaluation approaches do not define a fixed set of measurements that are applicable to all software models. Such sets of predefined measurements must be reduced or extended for different contexts and information needs. For the definition of an adequate measurement set, a systematic process is needed that guides modeling teams based on a model's context factors.
3. *Common quality understanding is missing:* In a modeling team, every team member should have the same understanding of quality related to their development context. Otherwise team members interpret qualities like testability and complexity differently. False interpretations can lead to misunderstandings and to faulty development results. Therefore, we need definitions for related qualities for a consistent and common quality understanding.
4. *Distinction between objective and subjective measurements is missing:* Objective measurements are more efficient, but some quality problems require the human judgement. Hence, we need a distinction between objective and subjective measurements.

The shortcomings of existing testability measurements and evaluation techniques listed above show that we need more comprehensive and systematic techniques for evaluation of model quality in general. Quality models [ISO9126] and the Goal Question Metric (GQM) [BCR94] are approaches that can help in solving these shortcomings.

Quality models are a state of the art technique for describing *what is measured* in order to evaluate a model [ISO9126]. Using quality models has the following advantages: At first, quality models enable the definition of a common quality understanding. Secondly, they enable the operationalization of model quality by objective and subjective measurements. These advantages can help in solving the shortcomings 3 and 4 listed above. However context factors are missing in quality models and a systematic process is not defined for developing and extending quality models.

To tackle these weaknesses of quality models the GQM approach seems to be appropriate, because GQM considers the characterization of context factors for the organization and development project and provides a systematic process for setting up a measurement plan from scratch. With the advantages of GQM also the shortcomings 1 and 2 can be handled.

Using only quality models or GQM would not help in solving all shortcomings of evaluating model quality listed above. Thus, a combination of advantages of both quali-

ty models and GQM is needed for a comprehensive and systematic evaluation of testability of software models.

In previous research, we introduced a novel approach named Model Quality Plan (MQP) to combine the advantages of both GQM and quality models [VE08]. The MQP process starts with a characterization of context factors. Compared to context characterization in GQM, MQP optimizes this step for software models by adding further context factors and omitting irrelevant ones. For the definition of a quality understanding, MQP integrates quality models into the GQM process as a refinement step of information needs. The operationalization is facilitated by associating quality attributes as the most detailed level of a quality model with objective and subjective measurements.

In this paper, we will show by an example, how the MQP approach can be applied for evaluating testability of models. For that, we will show step by step how to integrate the testability checklists from [Bin99] into a MQP.

The rest of the paper is structured as follows: The next section gives a brief overview of the related work in this area and discusses advantages and disadvantages of the existing approaches. After that, section 3 presents the Model Quality Plan (MQP) approach. In section 4 we present our example and show how the MQP approach is applied to the checklists from [Bin99]. Section 5 closes with a summary and future work.

2 Related Work

2.1 Testability of UML models

Jungmayr defines testability as “the degree to which a software artifact facilitates testing in a given test context” [Jun99]. Thus the testability of UML models can be defined as the degree of their suitability for testing. A UML model can be represented by different diagrams like class diagrams, statecharts, collaboration diagrams, sequence diagrams etc. Binder explains how these different diagrams can be used in different testing activities [Bin99]. As follows, we give an overview on related work about measuring and evaluating testability of UML models.

Baudry et al. [BTS02] consider class diagrams and statecharts to be best suitable for testing activities since they model general aspects of a software system, while other diagrams only show snapshots of some partial behaviors of the system under consideration. They focus on class diagrams and propose an approach to measure class interactions as the basis for testing cost estimation. They measure inheritance complexity by considering class dependencies. Because of the formal definition of the quality attribute and the model under consideration, all measurements in [BTS02] can be done objectively and automatically. Based on the measurement results they make suggestions for improving testability. The intension of [BTS02] is just to measure the inheritance complexity of class diagrams as an indicator for testability. They do not provide a process for developing further measurements for testability considering other quality attributes or any subjective measurements.

Briand and Labiche suggest to specify testability requirements for different UML diagrams using stereotypes and OCL annotations [BL01]. Using OCL they specify sequential dependency constraints between use cases, object interaction constraints in collaboration diagrams, pre- and post-conditions and class invariants for class diagrams. We think that these requirements are useful for the context of TOTEM (Testing Object-orienTed systEms) project. However, it is open how this approach on incorporating testability requirements into models can be generalized. We want to enable a systematic process adaptable and applicable to arbitrary development projects to define testability requirements.

McGregor and Skyes define generic quality characteristics like correctness, completeness and consistency for testability of UML models in different phases of the development [MS01]. In a guided inspection, they manually examine the quality characteristics of different UML diagrams used in each development phase. For analyzing certain syntactic properties of the model and its elements they specify a checklist [MS01, p. 128]. The checklist items are very heterogeneous and are neither grouped by diagram types nor by development phase. As a result, by evaluating the checklist in a certain context, additional work has to be done to select the relevant checklist items. Also the relation of the checklist to the quality characteristics is not explained. Our approach facilitates organizing measurements with respect to quality characteristics and a certain context.

Binder shows in [Bin99] how statecharts can be used for test case generation for state-based testing of object-oriented systems. He defines which structural, syntactical and semantical requirements statecharts have to fulfill if they are used for test case generation. He organizes these requirements in five checklists that developers and testers should use for inspections. Compared to the checklist in [MS01], the checklists in [Bin99] are very detailed. The level of detail of the checklists in [Bin99] can also be seen as their weakness, if a modeler does not know when to apply which checklist item. Even if the testability requirements are grouped in five checklists, some checklist items may not be relevant for any development context. Also it is not clear how to extend the checklists with new checklist items systematically. Our approach solves these shortcomings by enabling configuring measurements with respect to the context and arbitrary quality characteristics.

Systematic measurement of testability is also handled by Jungmayr in [Jun02]. Using GQM, he systematically defines metrics for measuring test-critical dependencies of software components. The raw data for computing metrics is collected by the ImproveT tool from classes and packages. In this approach, in order to measure testability source code must be available. We aim at measuring testability on model level far before the source code is implemented so that problems with testability can be solved in earlier phases of the development process.

As the examination of the related work for measurement of testability shows, existing approaches do not address all of the shortcomings for testability measurement we mentioned in Sec. 1, namely 1) context factors, 2) systematic process, 3) common quali-

ty understanding, and 4) distinction between objective and subjective measurements. In the next sections, we explain how these shortcomings are handled by the quality approaches GQM and quality models.

2.2 Quality Models

In ISO/IEC 9126-1 it is specified that a "quality model should be used (…) for software products and intermediate products" [ISO9126]. According to this common standard, quality models are understood as a state of the art technique for describing *what is measured* in order to evaluate (intermediate) products as for example a software model.

For evaluating the quality of software models, a couple of authors propose using quality models [BCN92, KLS95, Moo98, SR98 and Rei02]. The structure of quality models follows always the same principle. Quality models divide the term *quality* into its essential *quality characteristics*. Each of these characteristics can be subdivided into more detailed *quality subcharacteristics* or finally into *quality attributes*. In contrast to characteristics, attributes are atomic properties of a software model that do not depend on any other attribute such that attributes can be measured directly. Attributes cannot be refined further and depict the bottom layer of the quality model. Attributes are quantified by objective or subjective measurements.

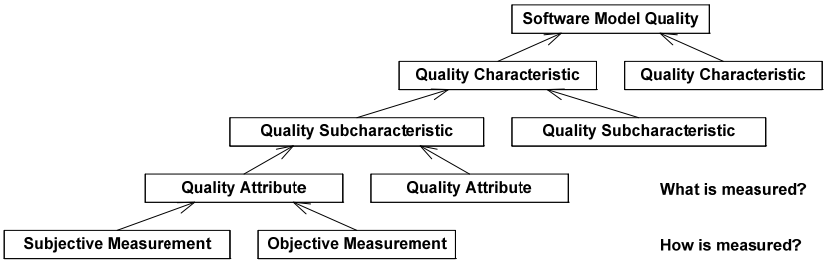

Fig. 1: Basic structure of a quality model

Therefore, quality models enable the definition of a common quality understanding and the operationalization of model quality by objective and subjective measurements (resolving the shortcomings 3 and 4 described in Sec. 1).

But quality models do not document their assumptions about the context characterization, so that it remains unclear if or to which degree a quality model is applicable to a given software model. In addition, there does not exist a systematic process for choosing, defining or extending a quality model based on context factors (shortcomings 1 and 2 remain unresolved).

2.3 Goal Question Metric (GQM)

To tackle these weaknesses the Goal Question Metric (GQM) [BCR94] is appropriate.

The first comprehensive foundations of the Goal Question Metric (GQM) have been established as a result of the TAME project (cp. [BR87, OB92]). Since that, the GQM

approach was extended by concepts (e.g. GQ(I)M in [PGF96], GQM++ in [GD97]), evaluated (e.g. [BSJ98]), formalized (e.g. [Dif93]) and best practices published (e.g [BDR97]). This diversity may be confusing. Preventive we present a minimal summary about the GQM approach that we talk about. We refer to the original GQM approach defined in [BR87, BO91, BCR94, and BDR97] as well as to the GQ(I)M described in [PGF96] that are predominantly the same. Representative for these publications we write shortened *GQM*.

Basically GQM consists of some main activities that are refined by several steps. We only concentrate on the GQM activity *Planning Measurement* (cp. Fig. 2:). At first, the *organization and the project context is characterized*. Somehow based on this given context, *information needs* are identified by documenting *goals* and related *questions*. For all questions subjective or objective *metrics* or *indicators* are defined. The result of this activity is a *GQM plan* that builds the basis for the execution of measurements.

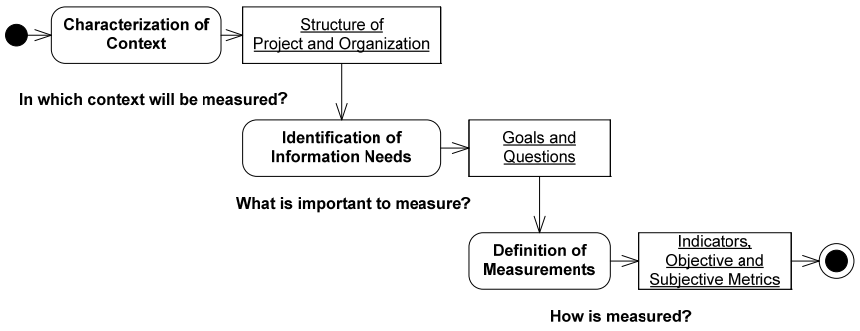

Fig. 2: UML activity diagram for the GQM activity "Planning Measurement"

GQM considers the characterization of context factors for the organization and development project. It provides a systematic process for setting up a measurement plan from scratch and differentiate between subjective and objective measurements (resolving the shortcomings 1, 2, and 4 described in Sec. 1).

But GQM does not explicitly focus on software models and does not provide all required context factors for them. Some context factors of GQM are not relevant for evaluating the quality of software models and these should be omitted (e.g. number of employees or potential customers). In addition, a quality understanding cannot be documented by the concepts provided by GQM (shortcomings 3 and 1, however partially, remain unresolved).

Using only quality models or GQM would not help in resolving all shortcomings of evaluating model quality listed in Sec. 1. Thus, a combination of advantages of both quality models and GQM is needed for a comprehensive and systematic evaluation of model quality. Thereby, the context characterization of GQM must be adopted.

3 Our approach

3.1 Specializing GQM for Software Models

Before introducing our approach, we first outline the most important changes in respect to GQM. We choose the comparison of our approach to GQM, because there is a strong conceptual overlap and GQM has a high degree of popularity.

Context of software models: GQM does not consider the context of software models to the whole extend. In GQM the context characterization includes the project and organization structure. The project structure is partially too coarse-grained. Therefore, we add some context factors that are special for software models like purpose of modeling or used diagram types. The organization structure is omitted, because we cannot confirm that it has any significant influence on the evaluation of software models.

Integration of quality models in GQM: We integrate quality models in GQM due to three reasons. First, we want to be able to define a common quality understanding, so that it is documented how important quality terms should be understood by the project team. For instance terms like maintainability, design quality, flexibility, and understandability are widely used. But these terms can be interpreted differently. Therefore, we need definitions for a consistent and common understanding about what quality means. Secondly, we want to specify what is measured. This means we need to name and define quality attributes of a software model that are quantified. Third, we would like to reuse existing quality models, because authors use them to describe their research results. But measurements are linked to questions in GQM and not to quality attributes as it is in quality models. Thus we must adopt the GQM approach.

GQM provides an anchor for the integration of quality models. The quality focus attached to goals in GQM is an undefined quality characteristic. In addition, in our approach questions are defined in respect to a quality focus, too. Thus, we make use of goals and question for deriving a first setup of a quality model.

Refining metrics to base and derived measures: We refine the concept metric to *base measures* and *derived measures.* There are many base measures as for example number of classes that are used as denominator in derived measures. These should not be multiple computed due to efficiency reasons. Furthermore, it is more comfortable to document base and derived measures in an enclosed part and to refer to these parts in other measures. This makes the definition of measures reusable, easier to read and understand.

Formalization of our approach: The aspects mentioned above influence the structure of our approach significantly. As a consequence, we cannot reuse available formalizations for GQM. Thus, we have to build a new one. We choose the concept of metamodel as an appropriate formalization, because the expected users of our approach are model experts. The UML is the de facto standard in the model based development and we assume that our users have UML knowledge and experience. On this account we define the structural and behavioral aspects of our approach by single and integrated UML model and provide views by UML diagrams.

3.2 Describing our Approach

Our quality management approach for the evaluation of software models is based on a top down process and a related metamodel. The process serves as a guideline for defining a Model Quality Plan (MQP) that describes how a software model should be evaluated. All relevant information contained in a MQP is typed and related to each other formalized by the metamodel. Hence, the metamodel states how a MQP may look like and the process guides one how to build up a MQP. Finally, the MQP can be applied to concrete software models (cp. Fig. 3:).

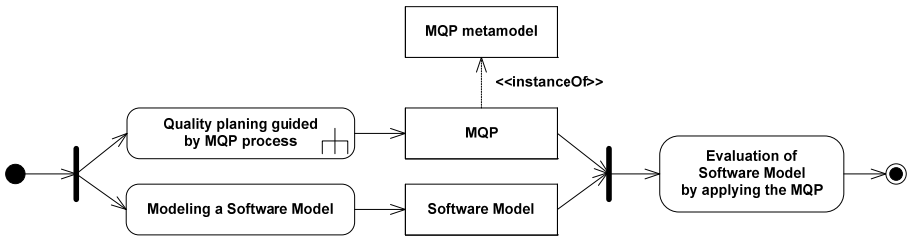

Fig. 3: Overview approach

Our MQP process is incremental and iterative:

1. At first, the context factors of a software model are documented in order to find out what is specific for the considered software model (e.g. used modeling language, diagram types, purposes of model and diagrams, development phase or development dependencies). In this step the question *in which context is measured* is answered, resolving the shortcoming 1 described in Sec. 1.
2. Secondly, the context factors are used for identifying information needs specified by goals and questions. Goals and questions are described with respect to quality characteristics and quality attributes respectively. These derived characteristics and attributes are the first setup for the quality model. During this step we try to figure out w*hat is important to measure*.
3. Third, the quality model is extended (e.g. subcharacteristics are introduced) and all characteristics and attributes are interrelated and defined. We substantiate the information needs, because we have to know *what is measured*, resolving the shortcoming 3 described in Sec. 1.
4. Forth, the measurement of the bottom level of the quality model (the quality attributes) is documented. We differentiate three measures: base measures, derived measures, and indicators. The documentation of a base measure includes for example a name, acronym, type and unit of measurement, informal or formal definition of the measurement method, scale and its scale type. Two types of measurement are distinguished. A measurement is subjective, if the quantification involves human judgment and objective, if it may be

implemented by automated means. The measurement definition tells us *how is measured*, resolving the shortcoming 4 described in Sec. 1.

The MQP process as a whole resolves the shortcoming 2 described in Sec. 1.

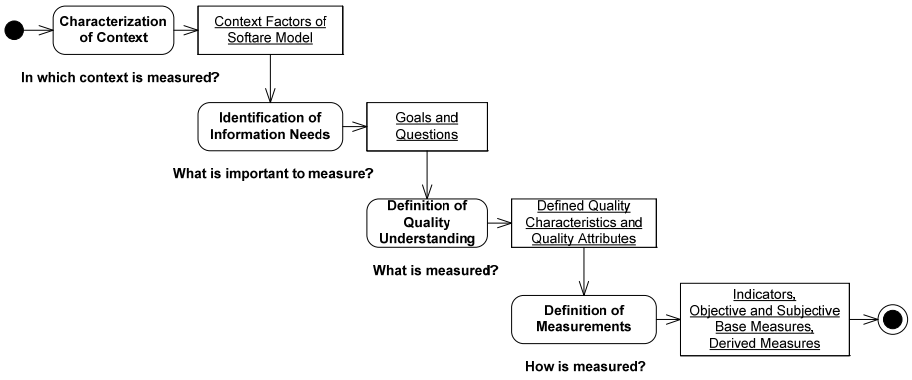

Fig. 4: Activity diagram - MQP process

The MQP metamodel groups classes according to the main steps of the MQP process by several packages (cp. Fig. 5:). The packaged classes possess a strong interrelated structure. The imported classes (e.g. ContextElement) point out the seamless transitions between two main steps. The intermediate result of one step is used for the following one.

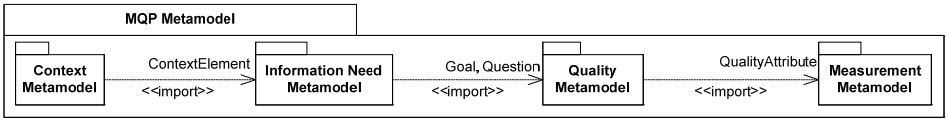

Fig. 5: Package Diagram - Extract of MQP metamodel

We present the strong interrelated structure of the MQP metamodel and the seamless transitions between two main steps by a small extract of the Information Need Metamodel (cp. Fig. 6:). The *ContextFactors DevelopmentPhase* and *DiagramType* are documented during the step *Characterization of the Context*. Such context factors are used for the *Identification of Information Needs*. Information needs are documented by *Goals* and refining *Questions*. Both are described with respect to a quality focus and build the first setup for the *Definition of Quality Understanding* by a quality model. Thus the quality model is based upon the intermediate results of the previous steps.

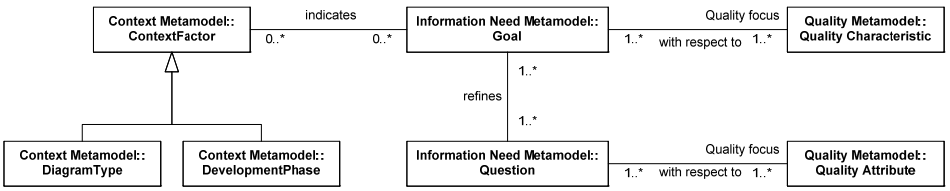

Fig. 6: Class diagram - Extract of Information Need Metamodel

4 Example for a Model Quality Plan (MQP)

Having introduced our approach, in this chapter we demonstrate how MQP works by a small example. We already introduced in Sec. 2.1 the testability checklists for statecharts in [Bind99] as a related work for measuring testability and addressed the shortcomings of checklists for measuring model quality in general. Now we show, how to incorporate these checklists into a MQP for testability measurement resolving the shortcomings of the checklists. In [Bin99] five checklist are defined containing 49 checklist items. For the sake of simplicity we will just consider 5 checklist items choosen one from each checklist.

Due to lack of space we use tables for the presentation of the MQP instead of object diagrams.

4.1 Context Model

The context of models is characterized by context factors in order to document their specifics and the application of the MQP (In which context is measured?)

Our MQP is applicable to models that are defined in *UML*, produced during the development phase *implementation design*, represented by *class diagrams and statecharts*, and used for the *specification of test cases* (Tab. 1:).

Tab. 1: Context model

Context Type	Context Factors
Modeling Language	UML
Development Phase	Implementation Design
Purpose of Model	Specification of Test Cases
Diagram Types	Class diagram and statechart

Information Need Model

Secondly, information needs are identified. The purpose of the Information Need Model is to document *what is important to measure.* The context factors are used for identifying information needs by goals and questions.

The goal is specified as follows: *Analyze the software model for the purpose of characterization with respect to testability in the context defined above (Tab. 1:).* This goal is refined by five questions (cp. Tab. 2:). Each question is described with respect to a quality attribute. The goal and its five refining questions already conclude the Information Need Model of our running example. The restricted Information Need Model suffices to demonstrate the advantages of our approach. Please consider that neither the goal nor the questions should depict an exhaustive picture, because we demonstrate in our example only a small part of the checklist items in [Bin99].

Tab. 2: Questions

Question	Quality Attributes
Is every state reachable from the initial state?	Dead States
Are the state names meaningful?	Meaningful State Names
Is each guard condition mutually exclusive of all other guards for a transition?	Nondeterministic behavior
Do subclasses consistently extend the statechart of their superclasses ?	Consistency of Redefined Statecharts
Is a timeout interval specified for each state?	Explicit Timeout Handling

Quality Model

Third step of the MQP process is to define and extend the quality characteristics and quality attributes by a quality model in order to specify the quality understanding and the properties of the software model that are measured (What is measured?).

The goal's quality focus *Testability* depicts the most upper layer of quality characteristics in the quality model (Fig. 7:). The bottom layer of the quality model is given by the quality attributes attached to the questions (Tab. 2:). We substantiate the dependencies between the upper layer and the bottom layer by introducing quality subcharacteristics. Dependencies are presented by arcs. An arc means that the source influences the target. An overview is given by Fig. 7: . The according definitions are contained in Tab. 3:.

It is clear that some important subcharacteristics of Testability like *Controlability* and *Observability* [Bin94] are missing in our example. The reason is that, by introducing the subcharacteristics, we only consider relevant subcharacteristics for the quality attributes of our example listed in the last subsection.

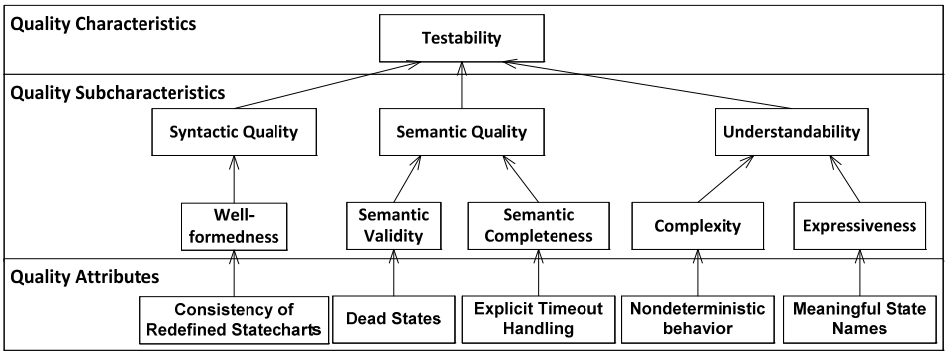

Fig. 7: Quality model

Tab. 3: Definitions of some quality characteristics and attributes for software models

Quality	**Definition**
Testability	The degree to which a software model facilitates test tasks in a given test context.
Syntactic Quality	Syntactic quality is the correspondence between the software model and its language definition.
Semantic Quality	Semantic quality is the correspondence between the software model and the domain.
Well-formedness	A software model is well-formed according to its language definition, if the software model complies to the constraints.
Semantic Validity	Validity means that all statements made in the software model are correct and relevant to the domain.
Semantic Completeness	A software model is complete when it represents all relevant features of the domain.
Understandability	The capability of the software model to be understood.
Complexity	The degree of difficulty in predicting the behavior and cooperation of a system defined by a software model.
Expressiveness	A software model is expressive when it can be easily understood through the meaning of UML model constructs, without the need for further explanation.

Meaningful State Names	A state name is meaningful if (1) it expresses relevant information in the application context (2) it is does not repeat the state variable invariant (3.1) in case of a simple state it has an active name (e.g. generating x, selecting x, …) (3.2) in case of composite state it has a passive name (e.g. hot, cold, on, off, high, low, …)
Consistent Extension of Redefined Statecharts	A redefining statechart is consistent with redefined statecharts if the redefining statechart (1) inherits all regions (2) does not remove states (3) does not remove transitions (4) does not weaken state invariants

Measurement Model

Now, we present examples for the subjective and objective measures for the quantification of the quality attributes. The intent of the Measurement Model is to formalize the determination of measures (How is measured?). Tab. 4: shows the assignment of a measure to two quality attributes. The descriptions of the measures are contained in Tab. 5 to 8.

The description of a measure is mainly structured as follows: A *Name* and an *Acronym* are used to refer to a measure. The *Measurement Method* specifies how to compute the measure. We differentiate the two *measurement types: subjective* (cp. Tab. 5) and *objective* (cp. Tab. 6). For both types of measurement, the measurement method is specified by an *informal definition* intuitively understandable for all project members. In case of an objective measurement, also a formal definition should be specified in order to enhance its automatic computing. Here, we use the Object Constraint Language (OCL) to give a formal definition for the measurement. The unit of measurement is especially important for subjective measurements, because they can be grouped to checklists that are used in inspections. The *scale* and its *scale type* defines possible measurement values. Base measures are functionally independent of other measures (cp. Tab. 5 and 7). In contrast, derived measures are defined as functions of two or more values of base measures (cp. Tab. 8).

Tab. 4: Some quality attributes and associated measures

Quality Attribute	Measure
Meaningful State Names	Ratio of meaningful state names
Consistent Extension of Redefined Statecharts	Number of subclasses that inconsistenly extend the statecharts of their superclasses

Tab. 5: Description of subjective base measure

Base Measure		
Name (Acronym)		Number of meaningful states (NMSt)
Measurement method	Informal definition	Count the absolute number of states that have meaningful names.
Type of measurement		Subjective
Unit of measurement		Statechart
Scale (Type of scale)		Integer from Zero to Infinity (Absolute)

Tab. 6: Description of objective base measure

Base Measure		
Name (Acronym)		Number of states (NSt)
Measurement method	Informal definition	Count the absolute number of states.
	Formal definition (OCL)	`context Class::NSt():Integer = if(self.Behavior.isOCLKind(StateMachine)) then self.Behavior.submachineState.size() else 0 endif`
Type of measurement		Objective
Scale (Type of scale)		Integer from Zero to Infinity (Absolute)

Tab. 7: Description of base measure

Base Measure		
Name (Acronym)		Number of subclasses that inconsistently extend the statecharts of their superclasses (NCIncSt)
Measurement method	Informal definition	Count the absolute number of classes that inconsistently extend the statecharts of their superclasses
	Formal definition (OCL)	`context Model::NCIncSt():Integer = self.allClasses() -> iterate (elem: Class; acc: Set(StateMachine)| if (elem.Behavior.isOCLKind(StateMachine)) then acc -> union(elem.Behavior) else acc endif) -> iterate (elem:StateMachine; acc: Integer = 0 | if (elem.isConsistentWith()) then acc + 1 else acc endif)`
Type of measurement		Objective
Scale (Type of scale)		Integer from Zero to Infinity (Absolute)

Tab. 8: Description of derived measure

Derived Measure		
Name (Acronym)		Ratio of meaningful state names (RMSt)
Measurement method	Informal definition	Divide the number of meaningful states by the number of states.
	Formal definition (OCL)	`context Class:: RMSt():Real` `= self.NMSt() / self.NSt()`
Scale (Type of scale)		Real from Zero to One (Ratio)

5 Summary and Future Work

In this paper, we focused on the evaluation of software models that are used for testing activities. We introduced the Model Quality Plan (MQP) approach for measuring quality of software models. We presented by an example, based on the testability checklists in [Bin99], how a MQP can be systematically developed for measuring the testability of UML statecharts.

The MQP approach is based on a combination of the Goal Question Metric (GQM) and quality models. It consists of a top down process and a related metamodel. The process guides one how to develop a Model Quality Plan (MQP) and the metamodel states how a MQP may look like.

The most important differences of our approach in respect to GQM are (1) adoption of the context characterization, (2) integration of quality models, (3) refinement of measurement level, and (4) the formalization of our approach by an integrated UML model.

Due to the documentation of a set of intermediate products and their high degree of details the initial effort for applying our approach remains heavy. Nevertheless, we are convinced that our approach is cost effective in the long term. Involving context factors tap the full potential to reuse existing quality plans. In addition, the systematic usage of context factors for the identification of information needs makes our approach more efficient. Last but not least the development effort can be considerably reduced by a dedicated tool support.

Our research is ongoing in the field of derivation rules for MQPs. Based on given context factors, we want to derive suggestions for MQPs that speed up their development. In addition, we will build up a set of comprehensive MQPs for different testing contexts.

References

[BCN92] C. Batini, S. Ceri, and S.B. Navathe: Conceptual Database Design: An Entity-Relationship Approach. Benjamin/Cummings, 1992

[BCR94] V. Basili, G. Caldiera, and H.D. Rombach: The Goal Question Metric Approach. Encyclopedia of Software Engineering, pp. 528-532, 1994

[BDR97] L.C. Briand, C.M. Differding, and H. D. Rombach: Practical Guidelines for Measurement-Based Process Improvement. Special issue of International Journal of Software Engineering & Knowledge Engineering, 2, 1997

[Bin94] R.V. Binder: Design for Testability in Object-Oriented Systems. Communications of the ACM, Vol. 37, No. 9, pp. 87-101, 1994

[Bin99] R.V. Binder: Testing Object-Oriented Systems - Models, Patterns, and Tools. Addison-Wesley, 1999

[BO91] V.R. Basili and M. Oivo: Representing software engineering models: The tame goal-oriented approach. Technical Report UMIACS-RE-91-155, CS-TR-2798, University of Maryland, 1991

[BR87] V.R. Basili and H.D. Rombach: TAME: Integrating Measurement into Software Environments, Technical Report CS-TR-1764, TAME-TR-1-1987, 1987

[BSJ98] A. Birk, R.v. Solingen, and J. Järvinen: Business Impact, Benefit, and Cost of Applying GQM in Industry: An In-Depth, Long-Term Investigation at Schlumberger RPS. In IEEE Metrics, pp. 93-96, 1998

[Dif93] C. Differding: Ein Objektmodell zur Unterstützung des GQM-Paradigmas [in German]. Masterthesis, Department of Computer Science, University of Kaiserslautern, 1993

[GD97] A. Gray and S.G. MacDonell: GQM++ a full life cycle framework for the development and implementation of software metric programs, 1997

[HL03] R. Heckel and M. Lohmann: Towards model-driven testing. Electr. Notes Theor. Comput. Sci. 82(6), 2003

[ISO9126] ISO/IEC: International Organization for Standardization (ISO) and International Electrotechnical Commision (IEC) 9126:2001: Software engineering Product quality Part 1: Quality model, 2001

[Jun99] S. Jungmayr: Reviewing software artifacts for testability. presented at EuroSTAR ’99, Barcelona, Spain, Nov. 8-12, 1999

[Jun02] S. Jungmayr: Testability measurement and software dependencies. In Proceedings of the 12th International Workshop on Software Measurement, October 7-9, 2002, Magdeburg, Germany, pp. 179-202, 2002

[KLS95] J. Krogstie, O.I. Lindland, and G. Sindre: Defining quality aspects for conceptual models. In IFIP Conference Proceedings, pp. 216-231. Chapman & Hall, 1995

[Moo98] D.L. Moody: Metrics for Evaluating the Quality of Entity Relationship Models. In Conceptual Modeling - ER ’98, 17th International Conference on Conceptual Modeling, Singapore, November 16-19, 1998, Proceedings, pp. 211-225. Springer, 1998

[OB92] M. Oivo and V.R. Basili: Representing Software Engineering Models: The TAME Goal Oriented Approach. IEEE Transactions on Software Engineering, 1992

[PGF96] R.E. Park, W.B. Goethert, and W.A. Florac: Goal-Driven Software Measurement - A Guidebook. Technical Report CMU/SEI-96-HB-002, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1996

[Rei02] R. Reißing: Bewertung der Qualität objektorientierter Entwürfe [in German]. Dissertation, Universität Stuttgart, Fakultät Informatik, 2002

[SR98] R. Schütte and T. Rotthowe: The Guidelines of Modelling – An Approach to Enhance the Quality in Information Models. 17th International Conference on Conceptual Modelling (ER ’98), Singapore, pp. 240-254, 1998

[VE08] H. Voigt and G. Engels: Kontextsensitive Qualitätsplanung für Software Modelle [in German]. In Modellierung, GI, LNI, 2008