An Overview of Quality Frameworks in Model-Driven Engineering and Observations on Transformation Quality

Parastoo Mohagheghi, Vegard Dehlen

SINTEF ICT, P.O.Box 124 Blindern,
N-0314 Oslo, Norway
{parastoo.mohagheghi, vegard.dehlen}@sintef.no

Abstract. Quality is often defined as fitness for purpose which is the key property to determine when evaluating quality. This paper presents some general requirements for evaluating quality frameworks. It also discusses characteristics of MDE that are important when building a quality framework, such as its use of models in several stages of development and maintenance, and its multi-abstraction level approach that requires consistency and traceability. MDE should strive for engineering of quality into software. As a step in extending quality frameworks for this purpose, we discuss integrating quality evaluation with quality engineering using MDE approaches. Applying it on transformations, we discuss both the transformability of models and the quality of transformations themselves. While completeness and precision of models are quality criteria for transformability of them, transformations should monitor and preserve model quality, be reusable, simple and efficient. We further discuss quality means and evaluation methods and propose future work.

Keywords: Model-driven engineering, quality, transformation, metrics

1 Introduction

More attention is paid to the quality aspects in Model-Driven Engineering (MDE) along with the growing importance of modeling in software development. Some challenging issues (especially for complex or large systems and special domains) are the increasing complexity that we need to understand and handle, the need for reliable systems and approaches that can verify and preserve quality requirements, as well as the dynamic adaptation and management of systems using transformations at run time. Our research on the "Quality in MDE" project in SINTEF tries to answer the following questions:

1. *What are the important quality aspects in MDE? Are there any differences in quality goals and activities when using MDE compared to other approaches?*
2. *How can quality goals be mapped into characteristics and metrics, and be evaluated?*
3. *How can quality be engineered into software systems by MDE?*

4. What experience do we have in evaluating quality in MDE?

This paper gives some answers to the first and second questions and defines an initial framework for engineering and evaluating quality in MDE. It further discusses how transformations can be used to answer the third question. We have performed a review of papers on quality aspects in MDE which allows us to summarize their findings and identify where there is need for further research. The literature on transformations quality is used here as an example to introduce the concepts.

The paper is organized as follows. Section 2 presents the different purposes of modeling, requirements on quality frameworks and characteristics of MDE that are important when defining a quality framework. Section 3 defines our quality framework that integrates model-based quality evaluation with model-based quality engineering and Section 4 applies it on the transformation quality. The paper is concluded in Section 5.

2 Background

2.1 Definitions of Quality and Relation to Modeling Purposes

Quality is often defined as "fitness for purpose". Thus the purpose of software decides what quality aspects are important in a given context. According to IEEE, software quality as an attribute is (1) the degree to which a system, component, or process meets specified requirements, and (2) the degree to which a system, component, or process meets customer or user needs or expectations [1]. ISO/IEC 14598 International Standard (Standard for Information technology - Software product evaluation - Part 1: General overview) defines the term *quality model* as "the set of characteristics and relationships between them, which provides the basis for specifying quality requirements and evaluating quality". *Model-driven Quality Assurance* (MDQA) is often defined as the automatic quality assurance that is based on models such as using system models for testing and verification. In this paper, we suggest the notion of *Model-Driven Quality Engineering* (MDQE) meaning the building / engineering of quality into software based on the model-driven approaches.

Models are developed for various purposes. Kühne classifies models as being either *descriptive* (capture some knowledge; e.g. requirements or domain analysis) or *prescriptive* (aka specification models; used as blueprints of a possible or imaginary system) [2]. In other words, a model can exist later or earlier than its original. Hesse thinks that in the software engineering field, a model often plays a double role: describing a part of an application domain and prescribing a piece of software for that domain [3]. Daniels defines three kinds of models based on their purposes [5]:

- *Conceptual models* describe a situation of interest in the world, such as a business operation or factory process.
- *Specification models* define what a software system must do, the information it must hold, and the behavior it must exhibit. They assume an ideal computing platform.

- *Implementation models* describe how the software is implemented, considering all the computing environment's constraints and limitations.

Claxton and McDougall write that assessing the quality of anything – models included – has two parts. One comes from measuring the right things, in the right way, with the right yardsticks. But the heart of quality comes from the second aspect; judging something based on its intended function and purpose. So the search for quality starts by asking, "What's the purpose of a model?" [4]. However, it is not often straight forward to define quality goals and requirements for each purpose of modeling or aspect, because:

- Some quality goals are in conflict with one another. For example using the same modeling language for different models helps understanding and reduces the need for learning new languages. On the other hand, we want to use different modeling features in each model (for example, the implementation model has to take the programming environment into account [5]) and using the same modeling language might therefore not be appropriate. In MDE, we move from Computation Independent Models (CIMs) to Platform Independent Models (PIMs) and Platform Specific Models (PSMs), and these models may also need different modeling languages and quality criteria. Paige et al. mean that users may profit of using different languages for different purposes and combining them [7].
- Some quality goals cross cut models or activities. One of them is traceability between models. For example, if our conceptual model contains the concept of *customer*, our software will contain direct representations of customers, and our software customers will have similar attributes to their real-world counterparts (the example is from [5]). We want this correspondence because it improves traceability between requirements and code, and because it makes the software easier to understand.

Thus any research on quality in MDE should take into account the various quality goals and the dependencies or conflicts between them. In MDE, models are refined progressively and transformed to new models or code. In [6], it is discussed that the quality of models depends on the quality of modeling language(s) used, the quality of tools used for modeling and transformations, the knowledge of developers of the problem in hand and their experience of modeling languages and tools, the quality of the modeling processes and the quality assurance techniques applied to discover faults or weaknesses. We also add the quality of activities performed on the models such as transformations to the above list, and discuss it in more details throughout this paper.

2.2 What Characterizes Model-Driven Engineering?

The characteristics of MDE that are important when defining a quality framework are:

- *Use of models in several stages of software development*: Models are used from early development phases to testing, simulation and code generation. Models are often incomplete, imprecise and inconsistent early in the software development

life-cycle and get gradually more precise and complete. Models can be non-executable or executable (even early analysis models can be executable).

- *Models on different levels of abstraction*: Relations between these models are important when evaluating them for some quality characteristics. For example, refined models have additional classes and methods that can increase complexity metrics.
- *Models from different viewpoints*: Examples are structural models vs. behavioral models. This is a characteristic of e.g. UML and not necessarily all modeling languages. The multi-view and multi-abstraction level development approach means that each of the diagrams and abstraction levels may require specific quality goals and metrics. Lange describes this for the model size metrics that varies on various diagrams and abstraction levels [11].
- *Activities are performed on models by tools*: Models undergo transformations and refinements. Many activities have models as input, output, or both [8]. The quality of such activities can preserve, improve or reduce the quality of models. Model transformation is applied by tools and during a transformation, output models are supplied with information not present in the input model. Examples are domain-specific information or the *platform* concept during the PIM to PSM transformation. Models should therefore be complete and precise but not include unnecessary or redundant information [8].
- *Generation of code and other artifacts form models*: This means that evaluating models is more important in MDE than in traditional software development where the code was mostly evaluated for quality.

Mellor and Balcer refer to several challenging issues that inevitably arise from the multi-view and multi-notational approach of UML in MDE [9]:

- *Consistency*: the models of various views need to be syntactically and semantically compatible with each other (i.e., horizontal consistency).
- *Transformation and evolution*: a model must be semantically consistent with its refinements (i.e., vertical consistency).
- *Traceability*: a change in the model of a particular view leads to corresponding consistent changes in the models of other views.
- *Integration*: models of different views may need to be seamlessly integrated before software production.

Besides, there may be some overlapping information, for example a class may appear in several class diagrams.

2.3 Related Work on Quality Frameworks

In this section, we present some work on quality frameworks that either address the quality of models or quality in MDE, or may be directly used in building such a framework for MDE.

Moody et al. write that we should address the following research questions when evaluating a quality framework [10]:

- Does the framework provide a reliable and valid basis for evaluating the quality of information models?
- Is the framework likely to be adopted in practice? It is critical for research to have an impact on practice.

They add that to assess the validity of a quality framework, we need to address the following questions:

- *Completeness (sufficient):* Do the set of quality categories include all aspects of quality of information models? Are there any aspects that have been left out?
- *Parsimony (necessary):* Are all the quality categories necessary for evaluating information models? Are they all relevant determinants of information model quality?
- *Independence (orthogonal):* Are the quality categories independent of each other? Do they overlap?

Trendowicz and Punter discuss quality models for software product lines (note that a *quality model* is a set of quality characteristics and relations between them to evaluate the quality of something. We use the term *quality framework* to avoid any confusion between quality model and model quality). They write that a very important issue in quality modeling is the phases of the software lifecycle to which the quality model is applicable [12]. In essence, quality modeling is more effective the earlier it can start in the software lifecycle, and the more phases it embraces. The activities during development of a quality model or framework are shown in Figure 1. The definition of goals should be done iteratively and involve the stakeholders.

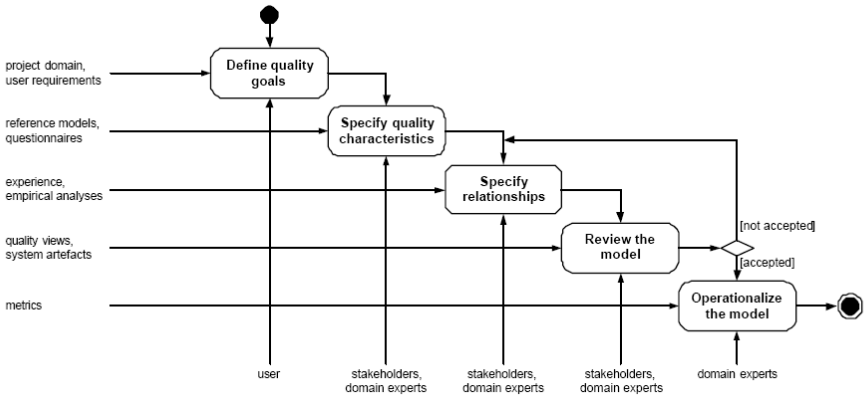

Fig. 1. Activities during development of quality models as defined in [12].

The authors list some characteristics for a quality model for software product lines that are also relevant for any quality framework:

- *Flexibility:* A quality model should be flexible because of the context dependency of software quality.

- *Reusability*: Depending on the projects' similarity level, a quality model should support the reuse of measurement data as well as quality characteristics and their relationships.
- *Transparency* - A quality model should provide the rationale of how certain characteristics are related to others and how to identify their sub-characteristics. Transparency of a quality model also means that the meaning of the characteristics and relationships between them are clearly (unambiguously) defined.

They further describe two kinds of approaches to model product quality: *fixed-model* and *define-your-own-model*. The fixed-model approaches lack flexibility and sometimes transparency, are often based heavily on metrics and do not include qualitative judgments. The define-your-own-model approaches can be adapted to the context and include qualitative judgments, but may result in poorly validated quality models.

A framework that is applied on conceptual models and evaluation of modeling languages is defined by Krogstie et al. (see for example [14]). The framework separates quality goals from means to achieve them. For example having formal syntax in a language is a means to achieve syntactic quality. Solheim and Neple have simplified and adapted this framework to MDE as depicted in Figure 2 from [8]. They further identify *transformability* and *maintainability* as two quality goals that are important in MDE, which are in turn decomposed to several characteristics. Their paper does not include empirical evaluation of the proposed characteristics.

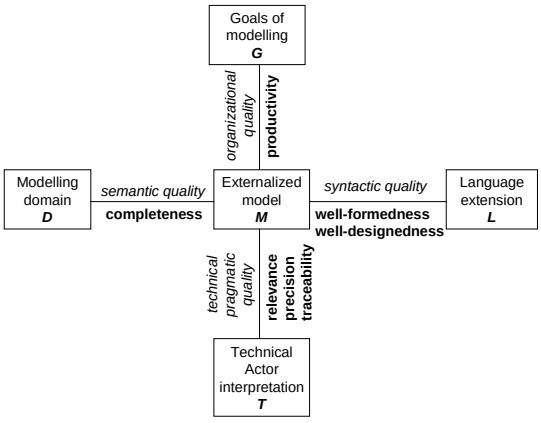

Fig. 2. Solheim and Neple's model for quality in MDE [8].

Lange and Chaudron identify two primary use of models; either development or maintenance. They further define some purposes of modeling for each phase (e.g., analysis and prediction are done in the development phase) and relate some quality characteristics to each purpose. These characteristics are further related to metrics that are mainly on the detailed design level [13]. This is a hierarchical model of software quality that we have found several examples of. Of other work on the quality of models we can mention [4] on the quality of data models (a data model is a model

describing parts of business) and [15] on the quality of UML 2.0 models (also provides some checklists for evaluating different UML diagrams for characteristics such as correctness and consistency).

In addition to models, modeling languages has been subject of research, as in [16], [14] and [7]. The three works have some language quality requirements in common such as having minimal set of concepts that are precisely defined, uniqueness of concepts and understandability, while they complement each other in other aspects. Another difference is when they are applied. Paige et al. [7] recommend their principles for designing modeling languages, while Krogstie et al. [14] and Grossman [16] have defined criteria for evaluating modeling languages.

Putting all the related work together provides requirements for quality frameworks and a list of quality goals and characteristics for some aspects such as models and languages, while other aspects such as processes, activities and tools are less studied. There is also a need for more empirical studies and evaluation of the frameworks.

3 Defining a Quality Framework for MDE

In the previous section we discussed requirements for quality frameworks, MDE characteristics, and some related work on the quality of models and modeling languages. In addition to defining quality goals and requirements for evaluating quality, we want to focus on the use of model-driven approaches in improving the quality of software. MDE lends itself to quality engineering because of two reasons. First, models are primary software artifacts in MDE and several other artifacts are generated from models. Thus developing high quality models improves the quality of e.g., test cases and code that may be fully or partly generated from models. Second, quality engineering is enhanced by the extensive use of tools in transforming models to other models or code. Tools can analyze and monitor models for various characteristics. An example is discussed by Haesen and Snoeck in relation with consistency checking which can be done *by analysis* (an algorithm detects inconsistencies between deliverables), *by monitoring* (meaning that a tool has a monitoring facility that checks every new specification), and *by construction or by generation* (meaning that a tool generates one deliverable from another and guarantees semantic consistency) [17]. Another example is using tools for checking some rules during modeling or transformations as proposed in [18], or adding constraints that should be verified before transformations. Rules, metrics and constraints can also be defined on metamodels.

We therefore modify Figure 1 to include both evaluating quality and engineering quality in MDE, and define the activities in the two processes as:

1. Specify the target object(s): target objects can be models, metamodels, languages, tools, activities etc. Models can be single or include multiple views / abstraction levels.
2. Define quality goals / characteristics based on:
    - Lifecycle phase (stages of development, maintenance or run-time).
    - Purpose of the object as discussed in Section 2.1.

- Isolated or in relation with other objects: it may be a need to integrate models / languages/ tools/ activities with other models / languages/ tools / activities, or they may need to exchange data. Integration may require consistency, portability, traceability, compatibility etc.
- Stakeholders involved (humans or tools; novices or experts).
- Scale of the tasks.
- Domain-specific (specific requirements such as reliability for some domains) or general.
- Lifetime (long-living or not): lifetime has impact on the need for training, documentation, or maintainability.

3. Specify relationships. An example is given in [19] where it is shown that some characteristics affect several goals, or may be in conflict with one another.
4. Specify how to evaluate characteristics (measuring quantitatively by metrics or subjective evaluation, evaluating by inspections using checklists, or by interviewing users) or identify means that should be provided to achieve the goals (rules, constraints, monitoring by tools etc.).
5. Review and evaluate the framework in practice for completeness, orthogonality, parsimony, reusability, flexibility, transparency, relevance and possibility to be adopted, as discussed in Section 2.3.
6. Operationalize: Operationalization is either measuring / evaluation of quality characteristics or implementation of quality means. The impact of means should also be evaluated.

These steps are shown in Figure 3.

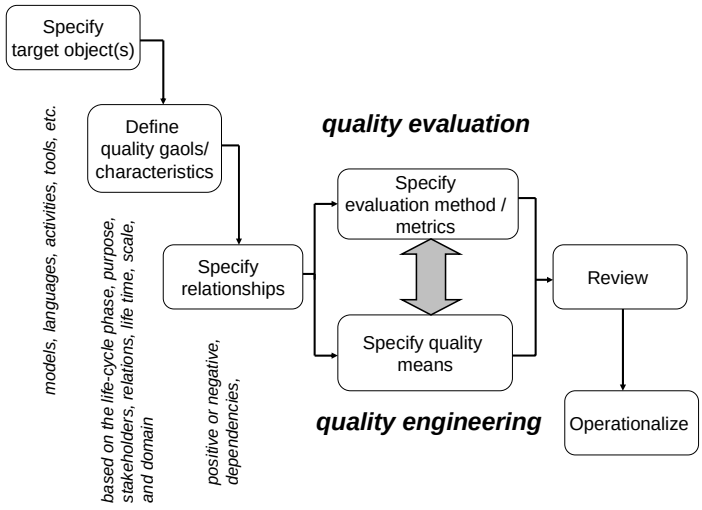

Fig. 3. The two aspects in a model-driven quality framework.

Sometimes a project has high level quality goals that may affect several objects. For example reusability of results may require paying attention to the reusability of models, change management, flexibility of transformations etc. Thus one might define

some high-level goals as well and refine them gradually to lower level goals related to specific target objects. In the next section we apply the approach to transformations.

4 Quality of Transformations

4.1 Motivation

A key point in MDE is the transformation of models. This technology has been proven useful both during the development and the maintenance of software systems, allowing refinements, new views or system code to be generated from models. The most useful transformations automate tasks that are either too tedious or complex for most developers to consistently and reliably implement [20]. But designing transformations, especially model-to-model transformations, is a complex task and should be considered as a project itself with clear requirements and verification steps (see for example [20]).

Kühne writes that a (model-to-model) transformation is information on a mapping from one model to another, created by a transformation engineer, for the transformation engine, in order to automate a transformation process [2]. So a transformation can be regarded as a model that describes a transformation function. Hesse, on the other hand, writes that although a transformation can be modeled if one wants to do so, the static model of a transformation should not be confused with its dynamic original [3]. In his view, transformations are processes and not models. In this paper, a transformation is a collection of transformation rules developed according to a transformation language and it may be regarded as a model.

Various works on transformations have emphasized different aspects of quality. Verró and Pataricza write that transformations should be regarded as models and their reusability, maintainability, performance and compactness should be considered [21]. Others have discussed that it is important that the output model maintains the properties of the input model; i.e., the transformation produces consistent models. We also have to consider in what way the transformation affects the quality of a model. This will of course depend on the metrics used; e.g. a refined model will usually be measured as more complex due to it having additional elements. Saeki and Kaiya, for example, present a simple example where the quality of two transformations is compared according to the value of a metric that reflects the amount of source code to be implemented – the less, the better [22].

Another important reason for considering the quality of transformations is that models and model transformations are increasingly being used at runtime. Self-adaptive systems have a runtime representation of themselves and adaptation occurs by manipulating this model. Changes to these models are subsequently transformed into running code. When using transformation at runtime, additional quality attributes come into play. In some systems, e.g. safety-critical ones, response times are usually important and, thus, the transformations have to adhere to constraints on timeliness. Also, during runtime adaptation it is even more important that the transformations maintain consistency and reliability among system configurations.

One of the key principles of software engineering is reuse, which also extends to transformations. Just like software components and services should be reused when building new systems, so should transformations be reused when developing new transformations. As transformations are developed and made available, however, developers need a way of determining which out of possibly several candidate transformation mappings to choose. A relevant example of a transformation repository is the ATL Transformation zoo, which is a part of the eclipse project[1]. Having access to quality criteria for transformations would allow meaningful comparison of transformation quality according to a set of chosen quality metrics.

4.2 Applying the Quality Framework to Transformation Quality

In this section we will provide a first iteration of the approach outlined in Section 3 on transformations, looking at quality from the evaluating and engineering points of view.

The first step in the process is to define the *target objects*. Several factors can play a role in the quality of a transformation in MDE:

1. The transformation engine (performed by a tool)
2. The transformation language and approach
3. The transformation rules
4. The source model
5. The source / target metamodels
6. The transformation tool

In this paper we focus on the four first objects since we do not have space to cover them all, leaving discussions of source and target metamodels as well as the tool quality for future work. Some work has already been done on the quality of languages / metamodels, as summarized in Section 2 and some quality criteria for transformation tools are presented in [27].

Next, we need to *formulate a set of quality goals* attached to the selected target objects, and select *means* for quality engineering or *metrics* for quality evaluation. Selecting relevant goals depends on the purpose of a transformation and other characteristics such as the size of models.

Transformation engine. Concerning the transformation engine, the most important quality criteria is its performance, which can determine whether or not a transformation is useful in a given scenario – especially at runtime. From the *quality evaluation* point of view (see Figure 3), defining characteristics for the performance of a transformation can be difficult, as it relies heavily on the implementation of the transformation engine / tool. It is possible to do quantitative measurements of the performance during the transformation process. However, the usefulness of this information is limited to a specific scenario between a set of fixed models. In addition, one would often want information about the performance of a transformation before it is used. Another alternative for measuring the performance is to gage the complexity of the transformation rules.

[1] http://www.eclipse.org/m2m/atl/atlTransformations/

From the *quality engineering* point of view, a means to achieve high performance is to select the appropriate transformation approach [25, 26]. In addition, important goals in software engineering are to support the traceability of requirements onto models and code and to support round-trip engineering. Both of the goals require that the transformation tool stores the transformation traces.

Transformation language. A transformation language should adhere to the design principles of a good language, as specified in Section 2.3. In addition, to allow reuse and simplicity through separation of concerns, the language should support the importing and/or inheritance (in the event of an object-oriented language) of other transformations. If a language is to be reused in multiple contexts, it should be flexible and maintainable.

Transformation rules. The limited literature that exists on the subject is mostly related to consistency during refinement. In [23], Liu et al. refer to several challenges that arise from the multi-view and multi-notational approach of UML – two being horizontal and vertical consistency during refinements. Several solutions are proposed in literature but we do not have space to cover them. Properties inherent to model refactoring can also be used as correctness properties in the context of model refinement, as done in [24]. Since refinements preserve certain correctness issues, model refinements imply the preservation of some behavior. Saeki and Kaiya define metrics for models and transformations [22]. They write: *"If a metric value can express the quality of a model, the changes of the metrics values before and after the transformation can be the improvement of the model quality."* To allow metrics to be calculated during transformations, the formal definition of a transformation should include the specification of metrics. Transformation rules can also include conditions that must be satisfied before the transformation is applied; for example checking completeness and precision of models.

Another goal is reusability, meaning that transformations should be subject to the separation of concern principle. A transformation should not become too big and do too many things. It is better to specify transformations that do specialized operations and rather chain several transformations to gain a greater effect. A reusable transformation will have to be generic in some way; however, this has to be considered against the tradeoff of being able to specify more detailed transformations and performance. Providing usable metrics for reusability is difficult, as the criterion is subjective of nature.

A key goal for the transformation rules is simplicity, which involves evaluating complexity. A metric that serves as an indicator of the complexity of a transformation is the number of rules. Also, we can identify two additional complexity criteria, namely: (1) the complexity of the rules and (2), in the case of an imperative transformation language, the complexity of the algorithms used in the mapping. As previously noted, the model complexity can also suggest something about the performance of the transformation process.

Czarnecki and Helsen have classified transformation approaches and offer some comments on the applicability of different approaches [25]. The aspects related to transformation rules are their logic, variables, patterns, being bi-directionality, additional control parameters, scope (allowing only part of a model being transformed), and rule organization / scheduling / application strategy. We related these aspects to some quality goals as shown in Table 1.

Models. In [8], the authors explored how we can measure the quality of models that are to be the subject of transformations, i.e. their *transformability*. Transformability is related to completeness (all elements are correct), relevance (all elements are necessary for a transformation), well-formedness (compliant with metamodel and profiles), and precision (being accurate and detailed for a particular transformation). Since there are approaches that do not include all elements of the source model in a transformation, relevance may be a subject of discussion. Czarnecki and Helsen have also discussed the relation between source and target models (target model may be a new model or an in-place update) [26]. It may be important for maintainability and evolution not to overwrite the source model.

The above discussion is summarized in Table 1. Table 1 is by no means a complete set of quality goals for transformations but it shows to some degree the state of work and the need to evaluate the impact of transformation approaches on quality goals. We also realized that some literature have defined approaches to transformation without discussing their impacts on quality goals, while others have focused on evaluating the quality without defining how to achieve it. Ideally, defining a quality goal should be accompanied by means and evaluation methods. There is no space to discuss relationships between goals or completeness of goals.

Table 1 An initial approach to defining a quality framework for transformations

Object	Goals	Means	Evaluation
Transformation engine	*High performance*	Effective transformation engine	Measure performance
		Select appropriate transformation approach [26]	
		Select scoping (including all elements or not) [26]	
	Round-trip engineering	Provide traceability (tool stores transformation traces)	
Transformation language	*Reusability*	Patterns [26]	
		Rule or module inheritance [26]	
	Maintainability	Generic transformations (may decrease performance) [22]	
	Ease of use	Similarity with known languages [27]	[27] Compares to QVT textual concrete syntax
	Flexibility	Hybrid approaches [26]	
	For model-to-text: see [27]	See [27]	
Transformation rules	*Preservation of properties*	For refinements: Observation / Invocation call	

Object	Goals	Means	Evaluation
		preservation by OCL constraints [25]	
		Correctness preservation [25]	
		Verify horizontal consistency	Analysis tool
		Verify vertical consistency	Analysis tool
		Enforce consistency by tools [18]	
	Preservation of quality		Measuring quality before and after transformation [23]
	Reusability	Specialized operations and rather chain several transformations to gain a greater effect	
	Simplicity	Few number of rules	Measure complexity in the number or size of rules, or the complexity of algorithms
	Verification and debugging of rules	Provide traceability	
Models	*Completeness [8]*	Include rules and constraints	Percentage [8]. Use tools for verification
	Well-formedness [8]	Include rules and constraints	Yes/no [8]. Use tools for verification
	Precision [8]	Check the model before transformation by tools	Yes/no [8]
	Evolution and impact analysis	Keep the source model after transformation	

5 Conclusion and Future Work

The MDE approach allows us to automate many activities in software development. Since models in MDE are expected to get progressively more complete, precise and executable and be used to generate the code and other artifacts such as test cases, they may be used to evaluate and verify the quality of design, fix errors and eliminate unwanted complexity. MDE also adds new requirements to the development process such as consistency between models and technical comprehension by tools.

Model-driven quality engineering and evaluation focus on integrating quality aspects into tools, modeling languages and activities such as transformations,

monitoring quality and evaluating it during the course of software development. It could be more effective the earlier it starts and more models / activities it covers. We used the literature on transformations to show examples of goals, means and evaluation methods, and integrated them together to show that engineering and evaluating quality need to go hand in hand. Putting the goals together allows analyzing them for dependencies and verifying the set for completeness and orthogonality in future work.

However, much work is still needed in all the stages defined in Figure 3. Suggestions for future work on transformations are further analysis of what affects the quality of transformations, how quality can be evaluated and how we can engineer models and transformations of high-quality. Especially important is the development of tool support for quality engineering, as tools are such an important part of MDE. This would support the operationalization part of the MDE quality framework. We will build further on the framework presented here to identify quality goals /means / and evaluation methods for other aspects that affect the quality of models and are relevant for our partners in the MODELPLEX project [28]. One of such aspects is identifying quality criteria for Domain-Specific Languages (DSLs) appropriate for modeling large and complex systems.

Acknowledgements. This research was done in the "Quality in Model-Driven Engineering" project at SINTEF ICT- Oslo, Norway.

References

1. IEEE 610.12 IEEE Standard Glossary of Software Engineering Terminology
2. Kühne, T.: Matters of (Meta-) Modeling. J. Softw Syst Model 5, pp. 369-385 (2006)
3. Hesse, W.: More Matters on (Meta-) Modeling: Remarks on Thomas Kühne's "Matters". J. Softw Syst Model, 5, pp. 387-394 (2006)
4. Claxton, J.C., McDougall, P.A.: Measuring the Quality of Models. In: The Data Administration Newsletter (TDAN.com), http://www.tdan.com/i014ht03.htm, visited on 22 June (2007)
5. Daniels, J.: Modeling with a sense of purpose. IEEE Softw. 19 (1), pp. 8-10 (2002)
6. Mohagheghi, P., Aagedal, J.Ø.: Evaluating Quality in Model-Driven Engineering. In: Workshop on Modeling in Software Engineering (MISE'07), In: Proc. of ICSE'07, 6. p (2007)
7. Paige, R.F., Ostroff, J.S., Brooke, P.J.: Principles for modeling language design. Info and Softw Tech. 42, pp. 665–675 (2000)
8. Solheim, I., Neple, T.: Model Quality in the Context of Model-Driven Development. In: 2nd International Workshop on Model-Driven Enterprise Information Systems (MDEIS'06), pp. 27-35 (2006)
9. Mellor S.J., Balcer, M.J.: Executable UML: a Foundation for Model-Driven Architecture. Addison-Wesley (2002)
10. Moody, D.L., Sindre, G., Brasethvik, T., Sølvberg, A.: Evaluating the Quality of Information Models: Empirical Testing of a Conceptual Model Quality Framework. In: 25th International Conference on Software Engineering (ICSE'03), pp.295 - 305 (2003)
11. Lange, C.F.J.: Model Size Matters. In: Workshop on Model Size Metrics at MoDELS'06, 5 p. (2006)

12. Trendowicz, A., Punter, T.: Quality Modeling for Software Product Lines. In: 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'03), 7 p. (2003)
13. Lange, C.F.J., Chaudron, M.R.V.: Managing Model Quality in UML-based Software Development. In: 13th Int'l Workshop on Software Technology and Engineering Practice (STEP'05), pp. 7-16 (2005)
14. Krogstie, J.: Evaluating UML Using a Generic Quality Framework. Chapter in UML and the Unified Process, Idea Group Publishing, pp. 1-22 (2003)
15. Unhelkar, B.: Verification and Validation for Quality of UML 2.0 Models, Wiley (2005)
16. Grossman, M., Aronson, J.E., McCarthy, R.V.: Does UML Make the Grade? Insights from the Software Development Community. Info and Softw Tech. 47, pp. 383–397 (2005)
17. Haesen, R., Snoeck, M.: Implementing Consistency Management Techniques for Conceptual Modeling. In: Third International Workshop, Consistency Problems in UML-based Software Development III – Understanding and Usage of Dependency Relationships, pp. 99-113 (2004)
18. Berenbach, B: Evaluation of Large, Complex UML Analysis and Design Models. In: 26th Int'l Conference on Software Engineering (ICSE'04), pp. 232- 241 (2004)
19. Bansiya, J., Davis, C.G.: A Hierarchical Model for Object-Oriented Design Quality Assessment. IEEE Trans. Soft Eng 28(1), pp. 4-17 (2002)
20. IBM, http://www.ibm.com/developerworks/rational/library/apr05/brown/index.html
21. Verró, D., Pataricza, A.: Generic and Meta-Transformations for Model Transformation Engineering, In UML 2004, pp. 290-304 (2004)
22. Saeki, M., Kaiya, H.: Model Metrics and Metrics of Model Transformations. In: The First Workshop on Quality in Modeling, pp. 31-45 (2006)
23. Liu, Z., Jifeng, H., Li, X., Chen, Y.: Consistency and Refinement of UML Models. In: Third International Workshop, Consistency Problems in UML-based Software Development III – Understanding and Usage of Dependency Relationships, pp. 23-40 (2004)
24. Straeten, R.: Formalizing Behaviour Preserving Dependencies in UML. In: Third International Workshop, Consistency Problems in UML-based Software Development III – Understanding and Usage of Dependency Relationships, pp. 71-82 (2004)
25. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: Workshop on Generative Techniques in the Context of Model-Driven Architecture at OOPSLA'03, 17 p.
26. Oldevik, J., Neple, T., Grønmo, R., Aagedal, J., Berre, A.J.: Toward Standardized Model to Text Transformations. In: First European Conference on Model Driven Architecture – Foundations and Applications, pp. 239-253 (2005)
27. Mens, T., Czarnecki, K., Van Gorp, P.: A Taxonomy of Model Transformations. http://drops.dagstuhl.de/opus/volltexte/2005/11/pdf/04101.SWM2.Paper.pdf (2005)
28. MODELPLEX project: https://www.modelplex.org/