# Metrics for Analyzing the Quality of Model Transformations

M.F. van Amstel[1], C.F.J. Lange[2], M.G.J. van den Brand[1]

[1] Department of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{M.F.v.Amstel|M.G.J.v.d.Brand}@tue.nl
[2] Federal Office for Information Technology
Barbarastraße 1, 50735 Cologne, Germany
mail@christian-lange.com

**Abstract.** Model transformations become increasingly important with the emergence of model driven engineering of, amongst others, object-oriented software systems. It is therefore necessary to define and evaluate the quality of model transformations. The goal of our research is to make the quality of model transformations measurable. This position paper presents the first results of this ongoing research. We present the quality attributes we have identified thus far and a set of metrics to assess these quality attributes.

## 1 Introduction

Model driven engineering (MDE) is a software engineering discipline that focuses on models for the development of software. MDE combines domain-specific modeling languages for modeling software systems and model transformations for synthesizing them [1]. Model transformations thus become more and more important. An example of a model transformation is adding getters and setters to a UML class diagram. Similar to other software engineering artifacts, model transformations have to be used by several developers, have to be changed according to changing requirements and should preferably be reused. Because of the prominent role of model transformations in today's and future software engineering, there is the need to define and assess their quality. Quality attributes such as modifiability, understandability and reusability need to be understood and defined in the context of MDE, i.e., for model transformations.

The goal of our research is to make the quality of model transformations measurable. We therefore start by defining the meaning of several quality attributes in the context of model transformations. We plan to do this by creating a quality model specific for model transformations (similar to the general software quality model described by Boehm et al. [2]). This model is a hierarchical decomposition of a number of quality attributes. We propose a set of metrics for assessing these quality attributes. Metrics have been studied extensively to assess the quality of (object-oriented) software [3,4,5] and software designs [6]. Some of the metrics

defined in earlier studies can be adapted such that they can be used to measure certain aspects of model transformations. We will also define new metrics that are specific for model transformations.

This paper presents the first results of our research on the quality of model transformations. In this paper we focus on model transformations created using the ASF+SDF [7,8] term rewriting system, but we expect that our techniques can be applied to model transformations created using different transformation engines as well. An example of a model transformation created using ASF+SDF can be found in [9]. In this paper we present the quality attributes we have identified thus far and a set of metrics to assess these quality attributes.

The remainder of this paper is structured as follows. Section 2 shortly explains the term rewriting system ASF+SDF. In Section 3 we describe the quality attributes we identified as applicable to model transformations. The metrics we propose to assess these quality attributes are described in Section 4. In Section 5 the metrics are related to the quality attributes. Section 6 contains the conclusions of our initial investigation and gives some directions for further research.

## 2 A Short Introduction to ASF+SDF

In this paper we consider quality attributes of model transformations defined using the term rewriting system ASF+SDF. One of the main applications of ASF+SDF is transformations between languages. These transformations are performed between languages specified in the syntax definition formalism SDF using conditional equations specified in the algebraic specification formalism ASF. The two main advantages of ASF+SDF are its modularity and the syntax-safety it guarantees. Syntax-safety in the context of model transformations means that every syntactically correct source model is transformed into a syntactically correct target model. It is impossible to transform syntactically incorrect source models using ASF+SDF.

A model transformation in ASF+SDF conceptually consists of multiple transformation functions. Transformation functions transform language elements from the source language into language elements of the target language. A transformation function is defined by signatures and equations. The signatures of a transformation function consist of the name of the transformation function, followed by a list of arguments and a return value. Signatures are defined in SDF. Apart from function signatures, SDF is also used to define variables. An example of a function signature and variable definition in SDF is depicted in Figure 1. Every signature can have equations which form the implementation of a transformation function. These equations have to conform to their signature. An equation can have zero or more conditions. These conditions can be used, amongst others, to assign values to variables. Equations are defined in ASF. An example of a function implementation with one condition (line 3) in ASF is depicted in Figure 2.

```
context-free syntax
  transform(Attribute) -> Return_value
variables
  "$Attribute"    -> Attribute
  "$Return_value" -> List[[Attribute]]
```

**Fig. 1.** Function signature and variable definition in SDF

```
equations
[transform-1]
  $Return_value := [$Attribute]
  ====>
  transform($Attribute) = $Return_value
```

**Fig. 2.** Function implementation in ASF (equations)

## 3 Quality Attributes

This section contains a description of the quality attributes that we have identified as relevant for model transformations thus far. Most of these quality attributes can be applied to software artifacts in general. Therefore we mention their relevance for model transformations in particular.

We plan to create a quality model specific for model transformations similar to Boehm's general software quality model described in [2].

*Understandability* The amount of effort required to understand a model transformation. Understandability is related to modifiability and reusability. The easier it is to understand a model transformation, the easier it is to modify or reuse. Since a model transformation is defined on a source and target (meta)model, their syntax and semantics should also be well understable to understand a model transformation.

*Modifiability* The extent to which a model transformation can be adapted to provide different or additional functionality. The main reason for modifying a model transformation is changing requirements. Another reason is that the (domain specific) language in which the source and/or target model are described may be subject to changes. Modifiability captures the amount of effort needed to modify a model transformation such as to deal with changes in either the requirements, or the source or target metamodel.

*Reusability* The extent to which (a part of) a model transformation can be reused by other model transformations. Reusability refers to as-is reuse. Therefore it is different from modifiability, which refers to modifying a model transformation. Reusability is especially relevant for model transformations when a source model has to be transformed into different target models, or vice versa.

*Reuse* Reuse is the counterpart of reusability. Reuse is the extent to which a model transformation reuses parts of other model transformations. We consider this as a quality attribute since it is good practice to reuse tested units.

MDE advocates reuse of models, i.e., a model is reused throughout the development process and different artifacts are generated from a source model by performing model transformations. Since model transformations in an MDE setting have the same source model as starting point, it is to be expected that the first few transformation steps in these model transformations are similar. So, reuse could be a measure for how well a model transformation adheres to the MDE paradigm.

*Modularity* The extent to which a model transformation is systematically structured. With systematically structured we mean that every module in a model transformation should have its own purpose. Modularity is related to reusability. If functionality is well spread over modules it is more likely that parts of it can be reused for other model transformations. Therefore, the size of transformation steps is also an important aspect of modularity.

*Completeness* The extent to which a model transformation is fully developed. A model transformation is complete if it transforms a source model into a target model according to its specifications, i.e., all functionality has been implemented. An incomplete model transformation will result in an incomplete target model or no target model at all.

*Consistency* The extent to which a model transformation contains no conflicting information. Boehm [2] distinguishes two types of consistency: internal consistency and external consistency. Internal consistency refers to the extent to which a model transformation contains uniform notation. Internal consistency is related to understandability. Internal inconsistency may lead to inconsistencies in the target model. External consistency refers to the extent to which a model transformation adheres to its specification.

*Conciseness* The extent to which a model transformation does not contain superfluous information. Examples of superfluous information are code clones or unnecessary function parameters.

## 4 Metrics

This section contains the metrics we have defined for assessing the quality attributes for model transformations created using ASF+SDF. In [10] metrics are defined for SDF. Those metrics are applicable to language definitions, but we will focus on model transformations. ASF has the characteristics of a functional language. Therefore we were able to adapt metrics for functional languages, like the ones defined in [11], such that they can be applied to model transformations.

### 4.1 Size Metrics

The size of a model transformation can be measured in various ways. An obvious size metric is the *number of lines of code*. However, different programming styles may require different counting techniques which can lead to different measurements [12]. Therefore we propose to measure the size of a model transformation by counting the number of transformation rules. For ASF+SDF transformations this results in the following metrics: *number of functions*, *number of signatures*, and *number of equations*. Note that the number of signatures does not have to be equal to the number of transformation functions since transformation functions in ASF+SDF can be overloaded, i.e., a transformation function can have multiple signatures each having different argument lists or return values.

A model transformation usually consists of a domain-specific part and a domain-independent part, i.e., library functions. The proposed size metrics can be adapted to measure the size of the domain-specific (or domain-independent) part of a model transformation only.

### 4.2 Function Metrics

The size of a transformation function can be measured in different ways as well. Section 2 states that a transformation function has one or more signatures and that every signature has one or more equations. The size of a transformation function can be expressed in terms of its number of signatures or equations. Also, the size of the equations, defined as the number of conditions, can be included. This leads to three different metrics for measuring the size of a transformation function: *number of signatures per function*, *number of equations per function*, and *number of equations plus number of conditions per function*.

A measurement for the complexity of a transformation function is the average number of values it takes as arguments and the number of values it returns. These metrics are known as *val-in* and *val-out*. Note that an ASF equation can return only one value, but this can be a tuple consisting of multiple values.

Transformation functions generally depend on other transformation functions to perform their task. The dependency of a transformation function $f$ on other transformation functions can be measured by counting the number of times function $f$ uses other functions. The dependency of transformation functions on a transformation function $f$ can be measured by counting the number of times function $f$ is used by other functions. These metrics are similar to *fan-out* and *fan-in* as they are used to measure dependencies between components of software architectures.

### 4.3 Module Metrics

One of the main benefits of ASF+SDF is that it allows the creation of model transformations in a modular way. One aspect of the modularity of a transformation is the *number of (library) modules* that comprise the transformation.

A large number of modules is no guarantee for an understandable model transformation. The modules should be balanced in terms of size and functionality. The *balance of a module* can be measured by comparing the number of functions, signatures, and equations defined in that module with the average over all modules. Actually, we measure unbalance in this way.

In a similar way as for functions the dependency of modules on other modules can be measured. The dependency of modules on a module $m$ can be measured by counting the *number of times module $m$ is imported by other modules*. The dependency of module $m$ on other modules can be measured by counting the *number of import declarations in module $m$*. Also, the fan-in and fan-out of a module can be measured. *Fan-in* is the number of times a function defined in module $m$ is used by another function that is not defined in module $m$. *Fan-out* is the number of times a function defined in a module $m$ uses a function that is not defined in module $m$. These metrics can be combined to measure the *complexity of the information flow* between modules as proposed in [13]:

$$\text{Information flow complexity}(M) = (\text{fan-in}(M) \times \text{fan-out}(M))^2.$$

In a similar way it is possible to combine the fan-in and fan-out metric of transformation functions to measure the information flow complexity of a function.

In general it is good practice to let every module of a model transformation have only one purpose, i.e., it should be concerned with one specific part of the transformation. This leads to a better balance of functionality among modules, and hence to a less complex model transformation. A module should thus contain one main transformation function and helper functions. We consider the function that is responsible for achieving the purpose of the module as the main transformation function. Note that helper functions are of course also transformation functions. If a module contains more than one main transformation function, the module should be split into multiple parts, each containing one main transformation function. Therefore, we propose to measure the *number of main functions per module*. This can be done by creating a call-graph of the module. A call-graph is a visual representation of the dependency of functions on each other. A vertex in the call-graph of a module represents a function defined in that module. A directed edge from vertex $a$ to vertex $b$ represents that the function represented by vertex $a$ uses the function represented by vertex $b$. A main function $f$ is a function which has only outgoing edges or incoming edges originating from $f$ itself in the call-graph. This metric can also help to identify obsolete functions. A main function that is not used by any function from another module as well could be an unused function.

ASF+SDF enables the creation of parameterized modules. A parameterized module is similar to a generic class in C++. Examples of parameterized modules in ASF+SDF are the container modules list and table. These are generic lists and tables that can be parameterized such that they can contain elements of any type. Parameterized modules increase reusability. Therefore, we propose to measure the *number of parameterized modules* in a model transformation.

### 4.4 Consistency Metrics

A transformation function is defined as a set of signatures and associated equations. It is possible that there is a signature for a transformation function, but that there are no equations. This can happen for example when the model transformation is still under development. To detect this type of inconsistency, we propose to measure the *number of signatures without equations*. The other way around, i.e., equations without signatures, will be detected by ASF+SDF itself and therefore we will not introduce a metric for this inconsistency.

Usually variables are defined in a `hiddens` section of an SDF file. This means that they can only be used within the same module. This implies that a variable needs to be redefined if it is to be used in other modules. This may lead to inconsistencies because the same variable name in one module can be related to a different type in another module, or vice versa. Therefore, it makes sense to measure the *number of different variable names per type*, the *number of different types per variable name*, and the *number of unused variables*. Note that it is possible in ASF+SDF to define an unlimited number of variables using the Kleene star (`*`). For example the variable definition `"var"[0-9]*` means that `var` can be postfixed with any number of digits, thus enabling the creation of an unlimited number of `var` variables. We consider a variable unused in a module if it is never used in the module it is defined in.

A transformation function can have multiple signatures. A possible reason for this is that the transformation function is defined on a supertype and that each of the signatures deals with a subtype. Since all these signatures and accompanying equations have a similar purpose, it is likely that code clones are present. If the number of code clones (per code clone) exceeds a certain threshold, it may be advisable to create a function that covers the functionality of the code clones. Therefore, we propose to measure the *number of code clones*.

A start-symbol defines a starting point of a transformation. During testing and debugging it is likely that only parts of a transformation are used. To be able to use only a part, a start-symbol has to be defined. If there is more than one start-symbol present in a transformation, this could either mean that it is a leftover of the testing and debugging phase or that the transformation can be used in different ways. We propose to measure the *number of start-symbols*.

## 5 Relating Metrics to Quality Attributes

In this section we will discuss the relation between the metrics derived for ASF+SDF model transformation and quality attributes. Table 1 summarizes the discussion by indicating the relation between metrics and quality attributes.

*Size* (lines 1–6 in Table 1.) Size has a negative effect on the understandability and modifiability of a model transformation. The larger a model transformation is, the harder it is to understand or modify.

The size of the domain-specific part of a model transformation has a negative effect on reusability and reuse. This part of a model transformation is specific for

a transformation and it is therefore unlikely that it can be reused for transformations or that parts from other transformations can be reused. It would however be interesting to look for similarities among model transformations that have the same source or target model. In this way reusability, and also reuse, can be assessed more accurately.

The size of the domain-independent part has a positive effect on reuse. The domain-independent part of a model transformation is defined as the part that consists of library functions. Since these functions are in a library, they are already being reused. It can also be the case that during the development of a model transformation certain transformation functions are generic enough to put them in a library. In this case the size of the domain-independent part of a model transformation has a positive effect on reusability.

*Function* (lines 7–13 in Table 1.) The size of functions has a negative effect on understandability and modifiability of a transformation. Moreover, the number of signatures and equations per function has a negative effect on consistency. If more similar signatures or equations have to be written, it is more likely that a different style is used.

A high value for val-in or val-out generally means that a function is specific. This has a negative effect on reusability. The number of input parameters and return values also has a negative effect on understandability and modifiability.

A high fan-in value means that a function is often used by other functions. This can be an indication that the function is generic, which benefits reusability.

A high fan-out value means that a function uses a lot of other functions, among which may be library functions. Therefore fan-out benefits reuse.

*Module* (lines 14–21 in Table 1.) The number of modules is a metric for measuring the modularity of model transformations, though not a very good one. It needs to be combined with the metrics (un)balance and number of main functions per module to get an impression of how well the functionality of a model transformation is divided over modules.

Functions are put in a library to be reused. Therefore, the number of library functions has a positive effect on reuse.

Similar to their variants for functions, fan-in and fan-out for modules also have a positive effect on respectively reusability and reuse. The combination of fan-in and fan-out, i.e., information complexity, is a measure of complexity. The more complex a model transformation, the harder it is to understand. Therefore this metric has a negative effect on understandability.

The purpose of a module with multiple main functions is unclear. Therefore the number of main functions per module has a negative effect on understandability. Modularity is also negatively influenced, since the module can be split into modules with only one main function. Because the module is less understandable and could be more fine-grained, it is less reusable.

Parameterized functions are created to be used in multiple forms. Therefore the number of parameterized functions has a positive effect on reusability. It

also has a positive effect on reuse, since container types like list and table are parameterized library functions.

| # | Metric | Quality Attributes | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Understandability | Modifiability | Reusability | Reuse | Modularity | Completeness | Consistency | Conciseness |
| **Size metrics** | | | | | | | | | |
| 1. | Lines of code | − | − | | | | | | |
| 2. | Number of functions | − | − | | | | | | |
| 3. | Number of signatures | − | − | | | | | | |
| 4. | Number of equations | − | − | | | | | | |
| 5. | Size of domain-specific part | | | − | − | | | | |
| 6. | Size of domain-independent part | | | + | + | | | | |
| **Function metrics** | | | | | | | | | |
| 7. | Number of signatures per function | − | − | | | | | − | |
| 8. | Number of equations per function | − | − | | | | | − | |
| 9. | Number of equations + conditions per function | − | − | | | | | | |
| 10. | Val-in | − | | − | | | | | |
| 11. | Val-out | − | | − | | | | | |
| 12. | Fan-in (function) | | | + | | | | | |
| 13. | Fan-out (function) | | | | + | | | | |
| **Module metrics** | | | | | | | | | |
| 14. | Number of modules | | | | | + | | | |
| 15. | Number of library modules | | | | + | | | | |
| 16. | Unbalance (module size − avg. module size) | | | | | − | | | |
| 17. | Fan-in (module) | | | + | | | | | |
| 18. | Fan-out (module) | | | | + | | | | |
| **Consistency metrics** | | | | | | | | | |
| 19. | Module information flow complexity | − | | | | | | | |
| 20. | Number of main functions per module | − | | | | − | − | | |
| 21. | Number of parameterized modules | | | + | + | | | | |
| 22. | Number of signatures without equations | | | − | | | − | − | |
| 23. | Number of variables per type | − | | | | | | − | |
| 24. | Number of types per variable | − | | | | | | − | |
| 25. | Number of unused variables | | | | | | − | − | |
| 26. | Number of code clones (per code clone) | | − | | | | | − | − |
| 27. | Number of start-symbols | | | | | | | − | |

**Table 1.** Metrics related to quality attributes

*Consistency* (lines 22–27 in Table 1.) The inconsistency metrics obviously all have a negative effect on consistency. Signatures without equations indicate that parts of the transformation are not finished yet. Therefore this metric has a negative effect on both completeness and reusability.

Variables with the same name but different types and variables with different names but the same type are confusing. Therefore the metrics referring to these inconsistencies have a negative effect on understandability.

Unused variables should be removed. Therefore the number of unused variables has a negative effect on completeness.

Code clones may be replaced by a function, such that the code has to be written only once. This would make a model transformation more concise. Therefore the number of code clones has a negative effect on conciseness. Also, if a part of the model transformation containing code clones has to be modified this has to be done in multiple places. Therefore this metric also has a negative effect on modifiability and consistency.

## 6 Conclusions and Future Work

In this paper we presented the first results of our ongoing research on the quality of model transformations. The main contributions is a set of eight quality attributes that can be used to assess the quality of model transformations. To refine these quality attributes and make them tangible, we have presented a set of metrics that can be used to assess these quality attributes. Our initial results presented in this position paper are a basis for future work in the direction of quality of model transformations.

To assess the quality of model transformations, first a clear definition of quality is needed. In Section 3 we presented the eight quality attributes we identified thus far. We plan to extend this set of quality attributes and relate them in a quality model such as proposed in [2].

In this paper we focused on ASF+SDF model transformations. We expect that our techniques can be generalized and applied to other model transformation formalisms, such as ATL [14] as well. The intended quality model will be the same, but some metrics to assess the quality attributes need to be adapted to the specifics of the transformation formalism. We proposed the metric number of functions as a measure for the size of a transformation created in ASF+SDF. For model transformations created with ATL the number of transformation rules could be used to measure size. However, we expect that most metrics will be conceptually the same for different transformation formalisms.

We want to verify our approach by means of empirical case studies. It is infeasible and inaccurate to extract metrics from model transformations by hand. Therefore we have to implement a tool that can automatically extract the values of all of the metrics from a model transformation. Furthermore we would like to visualize the values of metrics in such a way that outliers and striking values can easily be observed. Something similar has been done for software designs [15].

Once we have identified quality problems in model transformations, we can propose a methodology for improving their quality. This methodology will probably consist of a set of guidelines which, if adhered to, lead to high-quality model transformations.

**Acknowledgements** This work has been carried out as part of the FALCON project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute (BSIK03021) program.

## References

1. Schmidt, D.C.: Model-driven engineering. Computer **39**(2) (2006) 25–31
2. Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., Macleod, G.J., Merrit, M.J.: Characteristics of Software Quality. North-Holland (1978)
3. Rubey, R.J., Hartwick, R.D.: Quantitative measurement of program quality. In: Proc. of the 1968 23rd ACM national conference, ACM (1968) 671–677
4. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous & Practical Approach. 2nd edn. PWS Publishing Co. (1996)
5. Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity. Prentice Hall (1996)
6. Lange, C.F.J.: Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands (2007)
7. van den Brand, M.G.J., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF meta-environment: A component-based language development environment. In Wilhelm, R., ed.: Proc. of the 10th International Conference on Compiler Construction, Springer (2001) 365–370
8. van Deursen, A.: An overview of ASF+SDF. In van Deursen, A., Heering, J., Klint, P., eds.: Language Prototyping: An Algebraic Specification Approach. Volume 5. World Scientific Publishing (1996) 1–29
9. van Amstel, M.F., van den Brand, M.G.J., Protić, Z., Verhoeff, T.: Transforming process algebra models into UML state machines: Bridging a semantic gap? To appear in Proc. of the International Conference on Model Transformation (2008)
10. Alves, T., Visser, J.: SdfMetz: Extraction of metrics and graphs from syntax definitions. In Sloane, A., Johnstone, A., eds.: Proceedings of the 7th Workshop on Language Descriptions, Tools, and Applications. (2007) 97–104
11. Harrison, R.: Quantifying internal attributes of functional programs. Information and Software Technology **35**(10) (1993) 554–560
12. Jones, C.: Programmer Productivity. McGraw-Hill (1986)
13. Ince, D.C., Shepperd, M.J.: An empirical and theoretical analysis of an information flow-based system design metric. In Ghezzi, C., McDermid, J.A., eds.: Proc. of the 2nd European Software Engineering Conference, Springer (1989) 86–99
14. Jouault, F., Kurtev, I.: Transforming models with ATL. In Bruel, J.M., ed.: Satellite Events at the MoDELS 2005 Conference, Springer (2005) 128–138
15. Lange, C.F.J., Chaudron, M.R.V.: Supporting task-oriented modeling using interactive UML views. Journal of Visual Languages and Computing **18**(4) (2007)