Evaluating Maintainability with Code Metrics for Model-to-Model Transformations

Lucia Kapova, Thomas Goldschmidt, Steffen Becker, Jörg Henss

Chair for Software Design and Quality, Universität Karlsruhe (TH), 76131 Karlsruhe, Germany
{kapova, henss}@ipd.uka.de
FZI Forschungszentrum Informatik, 76131 Karlsruhe, Germany
{goldschmidt, sbecker}@fzi.de

Abstract. Using model-to-model transformations to generate analysis models or code from architecture models is sought to promote compliance and reuse of components. The maintainability of transformations is influenced by various characteristics - as with every programming language artifact. Code metrics are often used to estimate code maintainability. However, most of the established metrics do not apply to declarative transformation languages (such as QVT Relations) since they focus on imperative (e.g. object-oriented) coding styles. One way to characterize the maintainability of programs are code metrics. However, the vast majority of these metrics focus on imperative (e.g., object-oriented) coding styles and thus cannot be reused as-is for transformations written in declarative languages. In this paper we propose an initial set of quality metrics to evaluate transformations written in the declarative QVT Relations language. We apply the presented set of metrics to several reference transformations to demonstrate how to judge transformation maintainability based on our metrics.

1 Introduction

Model transformations are often used to transform software architectures into code or analysis models. Ideally, these transformations are written in special transformation languages like QVT [17]. With an observable increase in the application of Model-Driven Software Development (MDSD) in industry and research, more and more transformations are written by transformation engineers. Thus an increasing set of transformation scripts have to be maintained in the near future, i.e., they demand to be understood by other developers, bugs need to be tracked down and removed, and enhancements need to be implemented because of evolving source or target meta-models.

Today there are two main streams of model-to-model transformation languages: imperative (or operational) and functional (or relational) languages. For imperative languages like QVT Operational we can reuse existing literature about software code metrics for imperative, e.g. object oriented, languages. However, for relational model-transformation languages like QVT Relations there is not even a comparable amount of literature. In this paper we report on early experiences gained in our group on applying QVT Relations. They show that understanding relational transformations turns out to be quickly a difficult task. The difficulties increase more than linearly when transformation sizes increase and single relations become more complex.

In traditional object-oriented software development *software metrics* are used as a means to estimate the maintainability of code [2]. The estimated maintainability then indicates when the code base becomes too hard to maintain. Software developers take corrective actions like refactorings [7] or code reviews to keep the code in a maintainable state. However, these metrics do not yet exist for relational model transformation languages. Nevertheless, some initial research targets metrics for functional programming languages in general like Lisp or Haskell. Being part of the same language family, some metrics for functional programming languages can serve as a starting point for the definition of metrics for declarative model-transformation languages. In this paper we try to draw upon their ideas in defining our own set of metrics for model-transformation languages.

As an initial step towards estimating the maintainability of functional model transformation languages, we present a set of metrics usable to get insight into the maintainability of QVT Relations transformations. For this, we analysed existing metrics for functional programming languages and combined them with general code metrics (like Lines of Code (LOC)) and complemented them with our own experiences from applying QVT Relations. This set of developed metrics shall finally serve as a basis to judge internal transformation quality and to guide the development of transformation refactorings or review checklists (i.e., a list of bad smells to look for). We evaluated our metrics on the standard model-transformation example given by the QVT standard: the transformation from UML models to entity-relationship models to show that the metrics (a) are computable and (b) give insight into the transformation's internal quality.

The contribution of this paper are metrics to evaluate aspects of the maintainability of QVT Relational transformation scripts. These metrics are described in detail and their ranges of "bad" values are characterized including a rationale explaining which type of maintainability problem the metric detects. An early case study shows the metrics' applicability and initial evaluation results.

The paper is structured as follows. After discussing the properties of transformation languages in Section 2, we give an overview of related work to our approach in Section 3. Section 4 introduces identified quality metrics for transformations and Section 5 illustrates how to systematically compute the values for quality metrics. To demonstrate the applicability of our approach we introduce a case study in Section 6. We discuss the limitations and validity of the approach in Section 7. Finally, Section 8 concludes the paper and highlights future research directions.

2 The Group of Relational Transformation Languages

The goal of our work is to quantify the maintainability of model transformations. Therefore, we start by defining suitable metrics in this context. We identified a lack of quality metric definitions for relational transformation languages in the literature. Hence, in this paper, we focus on model transformations created using QVT Relational (QVT-R), but we assume that our metrics can be applied to model transformations created using other relational transformation languages as well. The main observed difference between relational and operational (i.e., imperative) languages is the fact, that operational

transformation languages describe a sequence of statements to create certain output. In contrast, relational transformation languages only describe the relations between input and output of a transformation in a declarative manner, not the way how it is computed (non-determinism). This results in special characteristics of relational transformation languages which have to be reflected by the metrics to be defined.

2.1 QVT Relational

QVT Relational is part of the QVT standard [17] and used for describing model transformations in a declarative manner. This means the transformation itself is written as a set of relations that shall be satisfied during the transformation process. As QVT Relational is multidirectional, there is no single source and target model but a list of so called candidate models. Each of these candidate models can be chosen as a target of the transformation, identifying the execution direction. When the transformation is invoked in a selected execution direction only the target model is modified such that all relations hold.

```
1  top relation ClassToTable {
2        cn : String;
3        prefix : String;
4        checkonly domain uml c : SimpleUML::UmlClass {
5          umlNamespace = p : SimpleUML::UmlPackage {},
6          umlKind = 'Persistent',
7          umlName = cn
8        };
9        enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
10         rdbmsSchema = s : SimpleRDBMS::RdbmsSchema { },
11         rdbmsName = cn,
12         rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn {
13              rdbmsName = cn + '_tid',
14              rdbmsType = 'NUMBER' },
15         rdbmsKey = k : SimpleRDBMS::RdbmsKey {
16              rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn{}}
17       };
18       when {
19       PackageToSchema(p, s);
20       }
21       where {
22         ClassToPkey(c, k);
23         prefix = cn;
24         AttributeToColumn(c, t, prefix);
25       }
26     }
```

Listing 1.1. Example of QVT Relational.

An example QVT-R transformation is given in Listing 1.1. A relation has two or more domains, that are given as patterns on the candidate models. The pattern usually

includes an object graph pattern, properties and associations between objects and defines a variable binding for each pattern match. By using the same variables in different domain patterns we can define the relation between candidate models. In consequence the target model is modified for each found pattern binding not being fulfilled to the extent that the relation holds. Beyond that, a relation can have *when* and *where* clauses that specify pre- and post-conditions. A relation only has to be satisfied when all precondition relations contained in the *when* clause are satisfied. In a similar manner each relation contained in the *where* clause has to be fulfilled when the relation containing the clause is fulfilled. Furthermore a target domain can be marked as *checkonly*, i.e. the target domain model is only checked for consistency and not modified. Besides this, relations are marked as *enforce* by default, thus insisting on the application of model changes for relations that do not hold. A relation can be marked as top-level. This means that the relation has to hold in any case for a successful transformation, while any non-top-level relation only has to be satisfied when directly or transitively referenced from a *where* clause.

2.2 General Observations on Maintainabilty of QVT-R Transformations

QVT-R can be applied for example in transformations between languages, code generation and incremental or refinement transformations. One main advantage of QVT-R is its brevity and conciseness. In the QVT-R language the structure of transformations is mainly characterised by the interdependencies of its relations. On the other hand relations can be defined in a way that they match overlapping sets of elements. Consequently, this increases complexity in cases when a new relation is introduced and it is influenced by other relations. For example, let transformation T be defined as a set of relations R, $R = \{a, b, c, d\}$. Suppose we want to extend T with a relation e, but e depends on a result of a and a depends on a result of both b and c, while c depends on d. Thus, we first need to understand how relations a, b, c and d are related in order to correctly include e into the transformation. In the case of more complex transformations it is very hard to have all dependencies in mind. Because of this net of dependencies it is hard to say if a new introduced relation conflicts with other relations or influences them in an undesired way. One possible design of relational transformation could be clustering of relations that match or create the same element (clustering of top-level relations). Furthermore, the identification of possible execution paths, how long they usually are and what they dependend on, is a very complex task.

3 Related Work

Quality metrics have been studied already to measure quality (software quality was defined by [3]) of object-oriented software [6, 12, 19], software architectures [1, 21] and design [15]. Metrics to estimate the maintainability of software are mostly based on measuring the size and complexity of code. Depending on the employed programming languages (functional, imperative, etc.) different metrics need to be employed for this task. The most relevant group of metrics for our approach is derived from related work

in the area of functional languages, such as the metrics defined by Harrison et al. in [11]. The group of relational transformation languages is related to functional programming languages, therefore we can reuse the existing functional metrics, similar to [22], in combination with some metrics used for object-oriented languages. However, Amstel et al. [22] focuses on model transformations created using the ASF+SDF transformation language. Most of these metrics are, however, quite generic and could be applied to nearly arbitrary functional programming languages. Nevertheless they do not take into account the special character of relational transformations, such as their strong alignment to the source and target metamodels. Still, some of these metrics can be used to measure certain aspects of model transformations written in QVT-R. We adapted some of the metrics to the special requirements of the QVT-R transformation language and extended them by the addition of more specific metrics (especially the group of manual metrics). Furthermore, we automated the gathering of the majority of the metrics presented in this paper.

In [8] initial considerations for transformation metrics based on a classification of transformation features [4] and a goal question metric plan were presented. However, these ideas were still in a very early stage and were not elaborated down to the special needs of different groups of transformation, such as relational transformations. Reynoso et al. [18] analysed how the complexity of OCL expressions impacts the analysability and understandability of UML models. As OCL is also part of QVT-R these findings are relevant for our approach. However, the remaining part of relational transformations, apart from OCL expressions, cannot be analysed using this approach. A special way of gathering a maintainability metric based on the occurrence of frequent patterns within a model or transformation was presented in [14]. The presented metric is based on a pattern mining approach that detects the most frequently occurring constructs. The assumption made in that paper is based on cognitive psychology, which says that the human brain works like a giant pattern matching machine and therefore can process things that re-occur often, more easily. Thus we incorporated this metric into our suite. Using OCL for the definition of metrics was introduced by Abreu in [5]. However, the approach presented there did not cope with metrics concerning the maintainability of transformations at all.

4 Metrics Definition

This section introduces metrics for measuring the quality of model transformations created using relational transformation language, such as QVT-R. For each metric we give a *description*, including a brief *motivation*. We also include the *rationale* behind the metric giving insights in why we believe the metric indicates the maintainability of a transformation. Additionally, we include a way for the *computation* (if possible using QVT-R and OCL) of the introduced metrics.

4.1 Automated metrics

In this section we will discuss the metrics derived for QVT-R that can be automatically computed. We identified four categories: Transformation Size metrics, Relational metrics, Consistency metrics and Inheritance metrics. In the following sections we will give

the names, descriptions and rationales of the automated metrics. Table 1 then gives the computation directions using OCL for the presented automated metrics.

Transformation Size metrics The size of the transformation has an impact on the understandability of a transformation. The size of a whole transformation can be measured in several ways. The number of *lines of code*, for instance, is a simple metric measuring the pure code size of a transformation. This is comparable to measuring lines of code in programming languages. Comments and blank lines are also included in this metric. The number of code, comment and blank lines can also be viewed separately. Used in conjunction with other metrics we can derive valuable measures of a transformation, e.g. when compared to the number of top level relations.

The *number of relations* is a metric that can be used to derive the degree of fragmentation and modularisation of a transformation. Higher number of relations can be considered better, as it is an indicator for a high degree of modularisation. A high degree of modularisation can support the maintainability of a transformation and also the reuse of a transformation or parts of it. The *number of top level relations* gives a picture about the independent parts of a transformation. A top level relation is a starting point for a transformation and can trigger the execution of other relations. An execution of a transformation requires all top level relations to hold. The ratio of top level relations to non-top level relations shows the rate between independent and dependent parts of a transformation. An interesting metric is *number of starts* defined by the number of top relations without when-clause. A higher number of starts increases the number of possible execution paths and therefore makes the transformation less maintainable. The metric *number of domains* expresses the complexity of a transformation dependent on the number of match patterns. The *number of domains predicates* additionally gives information about the complexity of these patterns. The *number of when-predicates* and the *number of where-predicates* defines how complex the dependency graph between relations is.

The *number of metamodels* in a transformation has an impact on the complexity of the transformation itself and its match patterns. The *size of the metamodel* (defined by a number of classes) on which the relations match elements might also have a great impact on the structure and therefore on the understandability and modifiability of the transformation. The larger the metamodel the larger the set of possible instances of this metamodel. Therefore, more combinations may have to be considered in the match patterns of the relations.

Relational metrics The size of a transformation relation can be measured in different ways. The OMG specification of QVT states that a relation has one or more domains and that every domain has a domain pattern that consists of a tree of template expressions. The size of a relation can be expressed in terms of its number of domains or the depth of the domain patterns. Additionally, relations can define when and where predicates giving pre- and postconditions. This leads to three different metrics for measuring the size of a relation: *Number of domains , Number of when/where predicates, Size of domain pattern per domain*. Another derived metric, the *ratio between the size of the relations and the number of relations* might also give hints about the maintainability of the transformation itself. However, the direction of the metric (e.g., for better maintain-

ability) remains to be evaluated. For example, having many but small relations helps to understand the transformation punctually, for specific relations. However, grasping the interconnections of many small relations is also a tedious and error-prone task, thus leading to the conclusion that having larger but fewer relations may be also good for maintainability. Still, defining a functional dependency between size and number of relations in a transformation might give hints on the maintainability of the transformation.

The metric *average number of local variables per relation* additionally gives indications on the dependencies within a relation that a developer needs to grasp when trying to understand and modify the relation. A measurement for the complexity of the interconnections between relations is the average number of arguments in the form of its domains and the number of variables that are bound by calls to other relations in when- or where- predicates. These metrics are denoted *val-in* and *val-out*. Note that in QVT-R *val-in* is always the same as *number of domains*. A high number of *val-out* means that a relation is strongly dependent on the context, which might decrease the reusability of a relation.

Relations generally depend on other relations to perform their task. The dependency of a relation R on other relations can be measured by counting the number of times relation R uses other relations or queries. These dependency metrics are denoted *fan-in* and *fan-out*, where *fan-in* is the number of calls to R and *fan-out* is the number of relations that are called by R. A high value of *fan-in* indicates that the relation is reused quite often and therefore is highly reused or somehow more central to the overall transformation. A high value of *fan-out* means that a relation uses a lot of other relations or functions (maybe delegates functionality to library queries), again making the relation more "central". The metric *number of enforce/checkonly domains* expresses a rate of change between the domains of the relation (e.g., source and target domain). The metric expresses the number of possible match patterns by the number of checkonly domains and the level of change provided by a relation (a number of diverse change patterns) by the number of enforce domains. The complexity of a transformation may furthermore be affected by the *number of OCL helpers* and *number of lines/restricted elements per OCL query*, which encapsulate more complex behaviour.

Consistency metrics A high degree of inconsistency in the transformation is a reason for confusion during development and may lead to reusability and transformation completeness problems. To detect an inconsistency in a transformation we introduce a number of consistency metrics. An example of inconsistency could be a relation that was not completed during development. Such a relation could be identified as a relation without domains, with only one domain or with domains without predicates. Therefore, we defined the metrics *number of relations without domains*, *number of relations with singular domains* and *number of domains without predicates*. An additional metric for the detection of incomplete relations is the *number of unused variables*. Unused variables pollute the code and complicate navigation within the transformation.

The already introduced consistency metrics are easy to automate. Another quite generic but still interesting metric is *number of clones*. However, the automation of this metric is a research field by itself. This metric identifies code duplicates, which are, as in other fields of code maintainability, candidates that impact maintainability of the code.

Inheritance metrics QVT-R transformations can extend each other and override relations from parents. Inheritance metrics measure the level of inheritance of the transformation and its complexity. The *balance* metric shows size and distribution of transformation functionality between children. This metric is calculated as a ration between a number of relations, domains and equations per child transformation in comparison to the average.

In a similar way as in object-oriented programming the dependency of children on their parents can be measured by counting the *number of transitive parents per child* and *number of direct/transitive children per parent*. Based on these metrics and the fan-in and fan-out metrics we can get a view of the dependencies between relations in the different transformations (create a dependency graph). The metric *number of overrides* gives information on how many relations from a parent transformation were overridden by a child relations. The larger this value gets, the more effort has to be invested into understanding which parts of the transformation hierarchy are actually used (combination of non-overridden (inherited), overridden and additional non-inherited parts).

4.2 Manually gathered metrics

In the following, we describe metrics that are not gathered fully automated but require manual or semi-automated analysis to determine the actual value of a metric.

Similarity of relations (frequent patterns) The *Similarity of relations (frequent patterns)* indicates how many similar patterns can be found in a transformation. A large part of the complexity of a transformation and on an model abstract model of the transformation comes through the need to understand patterns that occur within these models. The more complex a transformation is the harder it is to maintain it. Thus, to be able to grasp the complexity of transformations, we propose to emulate human information processing through pattern mining on models. Human analysis of software products is conducted either top-down or bottom-up according to [16]. Using a top-down approach the analyst tries to apply his/her knowledge about design and domain to classify the software product under analysis. In order to do this he/she tries to gain an overview of the whole application. He/She will then successively pick selected software segments and determine their relevance for his current mental model of the software. Using a bottom-up approach the analyst will start reading comments of source code or other software artifacts. The control flow of certain sections will then be inspected sequentially and arbitrary selected variables will be traced throughout the flow. Especially in declarative transformation languages this is a difficult task as there is no explicit control flow. The information gained will be integrated to a mental software model which is the opposite to the top-down approach. Masak [16] notes that top-down analysis is being conducted more often by experts whereas bottom-up analysis is being used more often by novice analysts. These findings give strong indication that experts may have abstract mental patterns at hand which are being used for analysing the software product whereas novices must resort to documentation. If analysability is measured in terms of time to analyse parts of a software product the required time will be low if the analysed parts dominantly adhere to the expert's patterns. On the other hand the time will be very high, if the expert can apply only a few of his/her patterns or the software

heavily differs from patterns known to him/her. These general observations were also stated for visual patterns in [20] which is why we propose to incorporate them into an analysability metric.

This metric can be computed by using the frequent pattern mining algorithm presented in [14] to identify possible frequent patterns. From these candidates the relevant patterns can be selected and their similarity can be estimated. However, the result of these pattern mining is mostly a superset of frequent patterns as they would be found by a human. Thus, manual selection needs to be performed to see whether each of the most frequent patterns is really a pattern that occurs as repeating structure in the transformation or if it is just the result of constraints on e.g., the transformation metamodel. For example, in QVT Relations a frequent pattern that is the result of the language concept would be that each relation domain has a root variable which refers to a meta-class that is contained in the package referred to by the domains typed model (see [17] for the QVT Relations metamodel). However, this construct in inherent to QVT relations and is not a frequent pattern that would be relevant for the analysability of a transformation. Thus, this metric cannot be computed fully automatically but needs an additional manual filter action. For example, a result of this metric could be that 30% of all relations of a transformation employ a pattern involving the matching or creation of a certain tree structure consisting of specific types of model elements within the source or target model. As humans are pretty good in pattern matching, a developer would then be able to recognise this combination over and over again thus helping him/her to more easily understand these 30% of relations.

Number of relations that follow a design pattern The *Number of relations that follow a design pattern* may be another important indicator for transformation maintainability. However, the determination of this metric is a tedious manual task as a design pattern is an abstract concept. It may occur in a form that can only vaguely be identified. The number of design patterns employed in the transformation may be a strong indicator on how good a transformation can be understood by external readers. However, as the area of transformation development is still quite immature only few design patterns have been identified yet. To determine this metric we need to count the number of design patterns and their occurrences within the transformation. For example, if a transformation uses the *Marker-Relation Pattern*[9] throughout its whole implementation and a developer knows what that pattern is used for he or she can grasp the meaning of the transformation more easily.

Type Cut Through Source/Target Metamodel The metric *Type Cut Through Source/-Target Metamodel* represents the rate of overlapping rules with respect to the transformation's metamodels. The type cut concerning a metamodel is the set of patterns that match instances of the same parts of a metamodel. In the UML to RDBMS example from the QVT standard (from which an excerpt in shown in Listing 1.1) the type cut concerning the meta-class UmlClass would be all those relations that contain a pattern that matches any UMLClass. The greater this overlap is, the more attention has to be paid when patterns of relations are modified in order to not lose coverage of possible instances of the metamodel.

To compute this metric we need to count the number of relations that overlap over the same part of a metamodel. For example, Relations a, b and c can all match instances

of the same meta-class m. Thus the overlap rate concerning class m would be 3. Finding type cuts that only refer to a certain element of the metamodel, such as one meta-class m can be done straight-forward. However, it might be more interesting more fine-grained patterns that are matched using several different relattions. How such a detailed type cut can be identified remains target to future research.

5 Computation of metrics

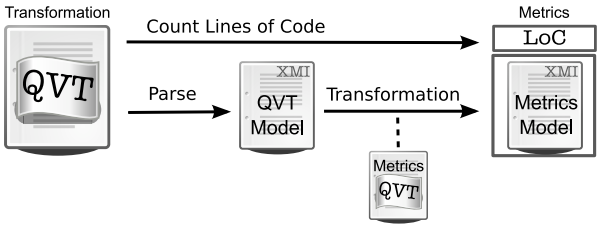

Fig. 1. Computation of metrics workflow.

The automated metrics described in section 4.1 can mostly be expressed as OCL expressions on the QVT-R meta-model. These OCL expressions can be used to count the number of elements of a specific type, for instance the number of relations a transformation has. The expressions have to be evaluated in the context of a transformation or a relation depending on wether a transformation local or relation-local metric is calculated. Table 1 shows the OCL expressions used for calculating the metrics. To bring these metrics together, relation local metrics can be aggregated by calculating an average.

```
1  query countSubExps(templ:QVTRelation::TemplateExp) : Integer
2  {
3  if (templ.oclIsTypeOf (QVTTemplate::ObjectTemplateExp))
4    then templ.oclAsType(QVTTemplate::ObjectTemplateExp).part->iterate(p:QVTRelation
         ::PropertyTemplateItem; acc:Integer = 1| acc + countSubExps(p.value.
         oclAsType(QVTRelation::TemplateExp)))
5    else
6     if (templ.oclIsTypeOf (QVTTemplate::CollectionTemplateExp))
7     then countSubExps(templ.oclIsTypeOf (QVTTemplate::CollectionTemplateExp).member
           .oclAsType(QVTRelation::TemplateExp)))
8     else
9     1
10    endif
11 endif
12 }
```

Listing 1.2. Query function for calculating the domain predicate count.

For more complex metrics like the domain pattern tree depth it was necessary to write more complex OCL query functions. Listing 1.2 shows an OCL query function

for recursively counting the nodes of a domain pattern tree. To easily apply all metric expressions and query functions, we developed a QVT-R transformation that transforms a QVT transformation to a special metrics model. The metrics metamodel allows for compact storage of metrics for every relation in a transformation and for the transformation itself. Moreover, it is possible to store the aggregated values that are also calculated by our metrics transformation. Furthermore, for measuring the lines of code we utilised common methods used for programming languages. We distinguished whitespace, pure comment and code lines. Figure 1 shows the workflow for retrieving the metrics.

6 Case study

In this section, we demonstrate how the introduced metrics give insight into the quality of transformations. We illustrate the applicability of our metrics generation approach and discuss the results. For this purpose, we present a case study based on an evaluation of three different transformations.

MOM (Message-oriented-Middleware) Completion Transformation This refinement transformation integrates performance-relevant details into software architectural models. These details are woven as additional subsystems into the model of architecture. The MOM completion transformation is dependent on the input from a mark model [4] that configures how the actual architecture model should be refined. The configuration, defined by the mark model, provides the variability to the transformation. For example, if a connector is to be refined by message-passing the mark model can provide information about the type of messaging channel, e.g., using guaranteed delivery. For further details on this transformation we refer to [13, 10]. Because this transformation is partially generated (includes copy relations for all metamodel elements, these relations are generated by the Ecore2Copy Transformation) we analyse this transformation twice: once with generated part and once without. The source and target model of this transformation are based on an underlying component-based metamodel with the size of 110 classes. This transformation is used as a representative of the group of quite complex transformations.

Ecore2Copy Transformation This transformation is a so called Higher-Order Transformation (HOT), as it generates another transformation. This specific HOT is used to generate a default copy transformation for a given metamodel by producing a copy relation for each class and each property of the given metamodel. This is required because there is no copy operator in QVT Relational. For further details on this transformation we refer to [9]. The source model of this transformation is the Ecore metamodel having 31 classes and target metamodel is the QVT Relations metamodel itself with the size of 110 classes. This transformation is used as a representative of the group of medium-complex transformations.

UML2RDBMS Transformation This transformation is presented in the QVT specification as an example relational transformation [17]. The UML2RDBMS transformation transforms UML class models into RDBMS tables. The minimum UML source metamodel contains 6 classes and the target RDBMS metamodel has a size of 18 classes. This transformation is used as a representative of the group of very simple transformations.

Name	OCL expression
Transformation t	
Number of relations	`t.rule → size()`
Number of top level relations	`t.rule → select(oclAsType(QVTRelation::Relation).isTopLevel) → size()`
Number of starts	`t.rule → select(oclAsType(QVTRelation::Relation).isTopLevel and oclAsType(qvtrelation::Relation).when → isEmpty()) → size()`
Number of when	`t.rule → iterate(r:qvtbase::Rule;sum:Integer = 0| sum + r.oclAsType(qvtrelation::Relation).when → size())`
Number of where	`t.rule → iterate(r:qvtbase::Rule;sum:Integer = 0| sum + r.oclAsType(qvtrelation::Relation).where → size())`
Number of metamodels	`t.modelParameter → size()`
Number of OCL queries	`t.ownedOperation → size()`
Relation r	
Number of domains	`r.domain → size()`
Number of enforced domains	`r.domain → select(isEnforcable) → size()`
Number of checkonly domains	`r.domain → select(isCheckable) → size()`
Number of when-predicates	`r.when.predicate → size()`
Number of where-predicates	`r.where.predicate → size()`
Number of local variables	`r.variable → reject(v | TemplateExp.allInstances().bindsTo.includes(v)) → size()`
Val-In	`see number of domains`
Val-Out	`Set{r.when} → including(r.where).predicate → collect( p | collectVariableArguments OfRelationCallExps(p)).variable → asSet() → size()`
Fan-In	`RelationCallExp.allInstances().referredRelation = r`
Fan-Out	`Set{r.when} → including(r.where).predicate → collect( p | collectRelationCallExps(p)) .referredRelation → asSet() → size()`

Table 1. Automated metrics

The results of this case study have shown that the generated transformation (GenMOMCompletion) in contrast to the transformation without the generated parts (MOMCompletion) has a higher number of small relations. Additionally, the complexity of match patterns is not high and the complexity of pattern matching is distributed on a number of relations (Figure 2). Thus, we see how the rate of domain pattern nodes per relation decreases significantly if the simple copy rules are added.

Transformation MOMCompletion, intuitively categorised as a complex transformation, shows a much higher values in average domain pattern tree depth as well as the average number of domains and when-predicates per relation(Figure 3). Interestingly, the number of where-predicates increases diametrically opposed. This may indicate that different approaches for defining the overall transformation have been employed. Moreover, where-predicates indicate a somehow "forward" (thus also more imperative) executed transformation whereas more when- predicates indicates a more declarative way of the whole transformation design. Which of these designs is more maintainable remains to be evaluated. However, using these metrics a connection between these findings could be underlined.

	GenMOM-Completion	MOM-Completion	Ecore2-copy	UML2 RDBMS
Lines of Code	7582	1304	473	239
Clean code	5789	1104	416	181
Comments	220	65	13	4
Number of relations	488	23	17	8
Number of top level relations	330	12	8	3
Number of starts	99	1	1	1
Number of OCL queries	20	21	1	1
Number of when-predicates	233	13	9	5
Number of where-predicates	221	5	12	13
Number of metamodels in transformation	3	3	3	2
Average number of domains per relation	2.11	4.652	2,76	2,5
Average number of domain pattern nodes per relation	2.63	14.78	11.529	2
Average number of when-predicates per relation	0.9	1.7826	1	0.63
Average number of where-predicates per relation	0.49	0.87	1.82	1.63
Average number of local variables per relation	0.001	0.478	1.05	2.375
Val-in per relation	2.63	14.78	11.529	2
Val-out per relation	2.3	4.45	3.66	3.12
Fan-in per relation	1.12	1.67	1.34	0.78
Fan-out per relation	1.02	1.34	1.2	0.7
Average number of checkonly domains per relation	1.04	2.09	0.714	1
Average number of enforce domains per relation	1.08	2.5652	2.47	1

Table 2. Automatically calculated metrics

The ratio between the number of top level relations and non-top level relations is the smallest in case of the generated transformation (1:1). This means a higher utilisation of top level relations. The generated transformation takes an advantage from a higher number of execution paths possible in the transformation and is not tuned to limit the number of starts in order to support maintainability. This also makes sense as the parts generated for the copy transformations are not intended to be maintained manually anyway.

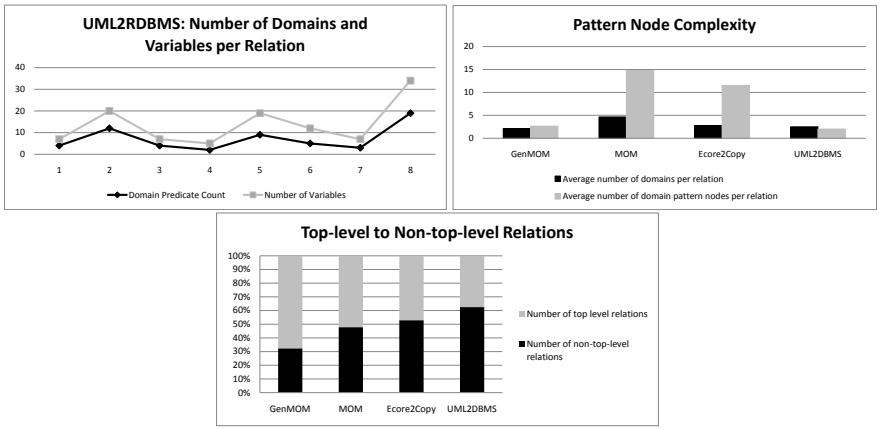

Fig. 2. Results: Transformation Complexity

In general, our observation is that roughly half of the relations are top-level relations. We can distinguish a pattern showing that a transformation was written manually by a human based on the number of starts as it seems natural for a human mind to consider only one execution path.

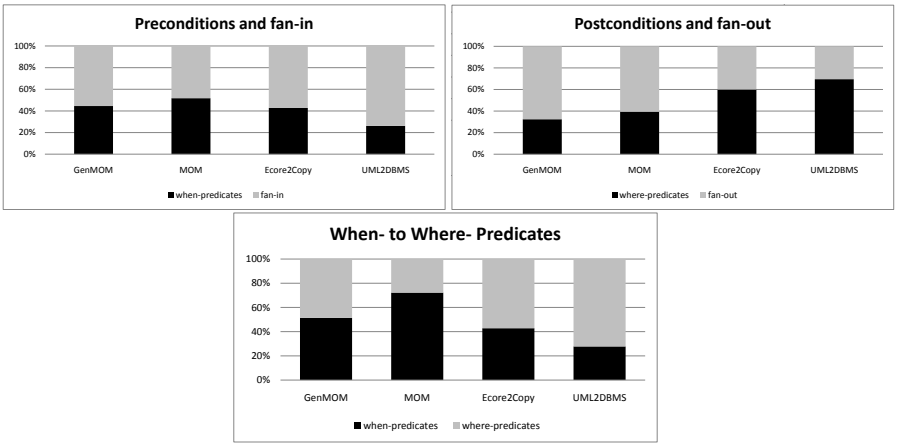

Fig. 3. Results: Relations Dependencies

7 Limitations and Validity

The definition of metrics with the goal to estimate quality attributes, such as maintainability, always comes with the wish to indicate whether a lower or a higher value of a metric is better or worse. However, this decision cannot be made without a sound validation of the "meaning" of a metric. For example, having a low number of relations, at first glance, seems to be good for maintainability whereas a high number seems to be bad. On the other hand, if these few relations are very long they may be harder to maintain that more but smaller relations. Thus, in this paper we only identified what could be possible indicators that may resemble maintainability of transformations. We intentionally did not decide, for most of our metrics, which "direction" of a metric is good or bad concerning maintainability. We leave it to future work to determine and evaluate this meaning. Thorough empirical evaluations need to be performed in order identify how meaningful each metric is.

8 Conclusions and Future Work

In this paper we presented an initial set of code metrics to evaluate the maintainability of QVT Relational transformations. However, such metrics could be applied to different relational transformations, they play important role when considering architecture refinement transformations. We demonstrated the use of these metrics on a set of reference transformations to show their application in real world settings. The presented metrics help software architects to judge the maintainability of their model transformations. Based on these judgments, software architects can take corrective actions (like refactorings or code-reviews) whenever they identify a decay in maintainability of their transformations. This results in higher agility when changing metamodels of software architectures or their platforms, which together with metamodel build basis for transformation definition. Future work is twofold. First, the identified metrics need to be incorporated into tools which indicate the code quality while developing the transformations in an IDE. Examples of such tools for object-oriented languages are Project Usus or Checkstyle . Second, the metrics must be empirically validated to study the extent to which they indicate decay in maintainability of transformations written in QVT Relational. Further some additional metrics could be identified as needed during this process, e.g. such as metrics for recursive relations and transformation cycles.

References

1. Steffen Becker. *Dependability Metrics*, volume 4909 of *Lecture Notes in Computer Science*, chapter Quality of Service Modeling Language, pages 43–47. Springer-Verlag Berlin Heidelberg, 2008.
2. Steffen Becker, Michael Hauck, Mircea Trifu, Klaus Krogmann, and Jan Kofroň. Reverse Engineering Component Models for Quality Predictions. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering, European Projects Track*, 2010.

3. B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
4. Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*. 2000.
5. Fernando Brito e Abreu. Using ocl to formalize object oriented metrics definitions. Technical report, FCT/UNL and INSC, 2001.
6. Norman E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman & Hall, Ltd., London, UK, UK, 1991.
7. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. 1999.
8. Thomas Goldschmidt and Jens Kuebler. Towards Evaluating Maintainability Within Model-Driven Environments. In *Software Engineering 2008, Workshop Modellgetriebene Softwarearchitektur - Evolution, Integration und Migration*, 2008.
9. Thomas Goldschmidt and Guido Wachsmuth. Refinement transformation support for QVT Relational transformations. In *3rd Workshop on Model Driven Software Engineering (MDSE 2008)*, 2008.
10. Jens Happe, Holger Friedrich, Steffen Becker, and Ralf H. Reussner. A Pattern-Based Performance Completion for Message-Oriented Middleware. In *Proceedings of the 7th International Workshop on Software and Performance (WOSP '08)*, pages 165–176, New York, NY, USA, 2008. ACM.
11. R Harrison, L. G. Samaraweera, M. R. Dobie, and P. H. Lewis. Estimating the quality of functional programs: an empirical investigation. *Information and Software Technology*, 37(12):701 – 707, 1995.
12. Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
13. Lucia Kapova and Steffen Becker. Systematic refinement of performance models for concurrent component-based systems. In *Proceedings of the Seventh International Workshop on Formal Engineering approches to Software Components and Architectures (FESCA 2010)*, Electronic Notes in Theoretical Computer Science, 2010.
14. Jens Kübler and Thomas Goldschmidt. A Pattern Mining Approach Using QVT. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA)*, Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2009.
15. C. F. J. Lange. Phd thesis: Assessing and improving the quality of modeling a series of empirical studies, 2007.
16. Dieter Masak. *Legacysoftware*. Springer, 2005.
17. Object Management Group. *MOF 2.0 Query/View/Transformation, version 1.0*, 2008.
18. L. Reynoso, M. Genero, M. Piattini, and E. Manso. Assessing the impact of coupling on the understandability and modifiability of ocl expressions within uml/ocl combined models. *Software Metrics, 2005. 11th IEEE International Symposium*, pages 10 pp.–, 19-22 Sept. 2005.
19. Raymond J. Rubey and R. Dean Hartwick. Quantitative measurement of program quality. In *Proceedings of the 1968 23rd ACM national conference*, pages 671–677, New York, NY, USA, 1968. ACM.
20. Robert L. Solso. *Cognitive Psychology*. Allyn and Bacon, 2001.
21. Johannes Stammel and Ralf Reussner. Kamp: Karlsruhe architectural maintainability prediction. In *Proceedings of the 1. Workshop des GI-Arbeitskreises Langlebige Softwaresysteme (L2S2): "Design for Future - Langlebige Softwaresysteme"*, pages 87–98, 2009.
22. M.F. van Amstel, C.F.J. Lange, and M.G.J. van den Brand. Metrics for analyzing the quality of model transformations. In *Theory and Practice of Model Transformations*, pages 239–248, 2009.