# Assessing and improving quality in QVTo model transformations

Gerpheide, C.M.

*Award date:*
2014

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Master's Thesis

# Assessing and Improving Quality in QVTo Model Transformations

Christine M. Gerpheide

*A thesis submitted in partial fulfillment of the requirements*
*for the degree of Master of Science in Computer Science and Engineering*

Supervisor: dr. A. Serebrenik, dr. ir. R.R.H Schiffelers

Examination Committee:
prof. dr. M.T. de Berg
prof. dr. M.G.J. van den Brand
dr. ir. R.R.H Schiffelers
dr. A. Serebrenik

Eindhoven, June 2014

*"Een duister raadsel is dit alles, hoe meer licht men erop tracht te werpen, des te meer men zich van de duisternis bewust wordt."*

Remco Campert
*Het Leven is Vurrukkulluk*

TECHNISCHE UNIVERSITEIT EINDHOVEN

# *Abstract*

Department of Mathematics and Computer Science

Master of Science in Computer Science and Engineering

**Assessing and Improving Quality in QVTo Model Transformations**

by Christine M. Gerpheide

We investigate quality in QVT Operational Mappings (QVTo), one of the languages defined in the OMG standard on model-to-model transformations. To do so, we pose two research questions. First, how can we assess quality of QVTo model transformations? For this we utilize a synthesis approach, combining bottom-up and top-down methods, including a broad exploratory study including QVTo expert interviews, a review of existing material, and introspection. QVTo transformation quality is then formalized in a QVTo quality model, consisting of high-level quality goals, quality attributes, and evaluation procedures. We evaluate the quality model by conducting a survey in which a broader group of QVTo developers rate each attribute on its importance to QVTo code quality. We find that although many quality attributes recognized as important for QVTo do have counterparts in traditional languages, a number are specific to QVTo or model transformation languages.

The second question we address is, how can we develop higher-quality QVTo transformations? Here we utilize the formulated QVTo quality model formulated to identify methods to help developers create QVTo transformations. Based on the identified methods, we chose to develop a test coverage tool for QVTo. The tool was implemented and then used by QVTo developers on real projects. After the usage period, the tool usefulness was evaluated by a conducting group interview, a review of usage data collected, and a tool performance analysis. The tool is available open source.

The primary contributions of this thesis are a QVTo quality model relevant to QVTo practitioners and an open-source code coverage tool for QVTo. Secondary contributions are a selection of QVTo best practices, a quality model evaluation approach leveraging developer perceptions, three code patches committed into the QVTo core engine which assist in the development of future tools, and numerous directions for future work in QVTo quality.

# *Acknowledgements*

I would firstly like to thank my advisors, Ramon and Alexander, for all their guidance and feedback during this project. Their input enabled this research to follow a sound, scientific path that I wouldn't have found without them. I'd especially like to thank Alexander for his lightning-fast feedback on paper and thesis revisions. I'd also like to thank the CARM2G team at ASML, who actively participated in parts of this research and provided active feedback on my project. Finally, I'd like to thank the Eclipse QVTo project maintainers for withstanding my questions about QVTo and all their advice making it possible to get patches accepted into the project.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **ASF+SDF** | **A**lgebraic **S**pecification **F**ormalism and **S**yntax **D**efinition **F**ormalism |
| **ATL** | **A**tlas **T**ransformation **L**anguage |
| **CARM** | ASML's **C**ontrol **A**rchitecture **R**eference **M**odel |
| **DSL** | **D**omain **S**pecific **L**anguage |
| **EMF** | **E**clipse **M**odelling **F**ramework |
| **GPL** | **G**eneral **P**urpose **L**anguage |
| **GQM** | **G**oal/**Q**uestion/**M**etric |
| **(J)VM** | (**J**ava) **V**irtual **M**achine |
| **MDE** | **M**odel-**D**riven **E**ngineering |
| **MMT** | **M**odel-to-**M**odel **T**ransformation |
| **OCL** | **O**bject **C**onstraint **L**anguage |
| **OOP** | **O**bject-**O**riented **P**rogramming |
| **QVT** | **Q**uery/**V**iew/**T**ransformation |
| **QVTo** | **QVT O**perational Mappings Language |
| **QVTr** | **QVT R**elations Language |
| **UML** | **U**nified **M**arkup **L**anguage |
| **URI** | **U**niform **R**esource **I**dentifier |

# Chapter 1

# Introduction

Model-driven engineering (MDE) can be used to develop highly-reliable software, offering benefits from system analysis and verification to code generation. In MDE, models of a system are created by domain experts and then *transformed* into other models or code using model transformations. One language for writing model transformations is QVT Operational Mappings, a.k.a. QVTo, specified in the 2007 Object Management Group (OMG) standard for model-to-model transformation (MMT) languages [7]. This standard represents consensus between a wide range of industries on what enterprise MMT languages should look like [8].

QVTo is a rich, imperative language, allowing developers to describe complex transformation structures [9]. Thanks to this richness and its presence inside the OMG standard for MMTs, QVTo is prominently used in both academia and industry [10–13]. In particular, ASML [14], the leading provider of complex lithography systems for the semiconductor industry, uses QVTo as their primary language for implementing MMTs. The ASML team cooperating with this research alone currently has approximately 20,000 lines of QVTo code supporting more than a hundred transformations.

However, QVTo is a relatively new language. While for general purpose languages (GPLs) developers have had time to build up many common notions to judge whether a piece of code is high- or low-quality, such shared best practices and quality indicators do not yet exist for QVTo. Moreover, QVTo has a large amount of language-specific constructs which hare not available in GPLs or even is other MMTs. In fact the QVTo specification has been described by some as "rather voluminous" and even "fantastically complex" [15]. Therefore, it is unclear whether intuitions about code quality for traditional languages apply to QVTo at all. This lack of standardized and codified best practices has already been identified as one of the largest current challenges in assessing model transformation quality [16].

Therefore in this thesis we investigate quality of QVTo transformations. We formalize our investigation in two research questions. The first addresses quality assessment:

***RQ1:*** *How do we assess quality of QVTo model transformations?*

This question seeks to assess or measure quality in QVTo. In addition to being able to assess quality, however, we also want to know how to use that information to improve the quality of QVTo transformations. Moreover, we want to know how to create

higher-quality transformations from the start, improving quality proactively. Therefore, we identify a second research question:

***RQ2:*** *How do we develop higher-quality QVTo model transformations?*

Whereas the first question concerns only assessment of a transformation, the second question also incorporates methods which lead to developing higher-quality transformations, for instance developer tooling or processes. Because of their large amount of QVTo code and increase that amount in the future, ASML in particular is very interested in these questions.

In Section 1.1 we discuss our scientific acceptance of truth by selecting a specific philosophical stance. We then make the contributions of this research in Section 1.2. Finally, we present the organization of the remainder of the thesis in Section 1.3.

## 1.1 Philosophical Stance

Before continuing, we make explicit the *philosophical stance* adopted in this research. Most importantly, this stance defines what one is willing to accept as empirical truth when answering our research questions. For example, one researcher may be believe that the only way to show that A increases B is to use have a controlled experiment with one group using A and one not and then measure B. However, a second researcher may believe that witnessing an increase in B in a lab setting does not mean it would be the same in the real-world. Therefore the second researcher may be more convinced if A increased when B increased in in the field, despite the presence of many uncontrolled variables. What evidence for truth is accepted is guided by which philosophical stance you adopt to approaching your problem. Easterbrook et al. [17] describes the stances and their connections to software engineering and empirical methods:

- **Positivism** states that knowledge is based on logical inference from observable facts. This is done largely by breaking up problems into small components which can be verifiably tested in isolation. Positivism is most closely related to controlled experiments. For example, measuring a code metric on on two pieces of code which are known to differ in exactly one aspect.
- **Constructivism** states that knowledge cannot be separated from its human context. Constructivists tend to construct *local theories* about why certain phenomenon occur which are coupled with their context. Constructivism is most closely related to ethnographies, i.e. a detailed qualitative description of a human practice, for example analyzing software processes to gain insight into what causes bugs.
- **Critical theory** states that knowledge is something that frees people from restrictive systems of thought. Critical theory is most closely related to action research, i.e. employing a concept in the field to judge its effect. In software engineering, it includes research that challenges existing software practices, such as the open source movement.
- **Pragmatism** states that knowledge is judged by how useful it is for solving practical problems. As pragmatisms suggests using whichever methods may work for a

given problem, it is most closely related to *mixed methods* research, i.e. utilizing a combination of methods. Notably, it is assumed in this stance that what may be most useful in one context may not be in another. For example, a specific development methodology may be the most effective for one software team but not for another, but is still judged by its usefulness within a specific context.

In addition to influencing the type of evidence accepted to prove a theory, these stances also play a large role in the practice of *theory building* from the start. For example, to a constructivist, theories emerge from the data collected, whereas to a pragmatist, theories are the products of a consensual process among researchers [17]. Although often left inexplicit, the method by which most researchers develop theories can be described as a relaxed form *grounded theory*. In grounded theory, some initial analyses are performed and patterns emerge. These patterns are used to refine the data collection iteratively until there is support of some theory. An elaboration on grounded theory is given by Glaser and Strauss [18].

In our research, we have chosen to maximize the relevance and usefulness of our contributions to industry. We therefore explicitly adopt the pragmatist philosophical stance. This stance drives many of our design decisions as well as our validation strategies as we address our research questions.

## 1.2 Contributions

The primary contributions of this thesis are twofold. First, a QVTo quality model relevant to QVTo practitioners is presented in response to our first research question. A paper [19] providing this contribution has already been accepted to the Quality in Model Driven Engineering track of QUATIC '14 [20]. It was also named one of the best papers in track and we have been invited to submit an extended version to the conference main track. Some of the reviewer feedback from the submission is presented in Chapter 8.

The second primary contribution is an open-source test coverage tool for QVTo which integrates into existing developer practices, developed while investigating our second research question.

This research also contributes a number of secondary contributions. First, a selection of QVTo best practices and difficulties gathered from expert interviews are presented. Second, our approach to designing and evaluating the quality model, unlike related work, leverages a synthesis approach and developer perceptions which we argue provides more convincing evidence of the quality model's practical relevance. Third, three code patches were committed into the QVTo core engine as a result of this research which will aid in the development of future QVTo tooling. Finally, we present many directions for future work in QVTo quality related to both research questions.

## 1.3 Thesis Organization

First we present preliminary information in Chapter 2, including background on model-driven engineering, the QVTo language, and the industrial context in which this research

was carried out. The remainder of the thesis is divided in two parts. Part I covers the formalization of QVTo quality, therefore primarily addressing the first research question. In Chapter 3 an introduction to software quality is given. In Chapter 4 we describe our approach to formalizing QVTo quality. In Chapter 5 we present intermediate results as well as the best practices and difficulties mention in Section 1.2. In Chapter 6 we present the formalized QVTo quality model. In Chapter 7 we present our evaluation approach and use it to evaluate our QVTo quality model. Finally in Chapter 8 we present conclusions and future work with respect to QVTo quality assessment.

In Part II the second research question is addressed. There we design, implement, and evaluate a tool developed based on the quality model from Part I. Our tool selection and design approach is presented in Chapter 9. It was chosen to implement a tool for QVTo code test coverage analysis. We formulate the specific requirements for the tool in Chapter 10. The tool implementation, as well as the patches contributed to the QVTo core, are described in Chapter 11. Then we evaluate the tool and discuss directions for future respect to tooling in Chapter 12. Finally, we conclude both parts of the thesis in Chapter 13.

Related work is discussed in respective chapters: Chapter 3 presents related work on software quality, which served as a starting point for this research. Chapter 4 contains an introduction to empirical methods research. In Chapter 5 the results from an in-depth, systematic literature review of quality are presented. In Chapter 7 we present related work on quality model evaluation techniques. We discuss related work in tool evaluation in Chapter 9. Finally, Chapter 10 presents related work on code coverage analysis.

# Chapter 2

# Preliminaries

In this chapter, preliminary material is presented. Section 2.1 provides an overview of model-driven engineering and model transformation languages. Section 2.2 gives an overview of the syntax and semantics QVT Operational Mappings (QVTo) language, the focus of this research. Finally, Section 2.3 describes on the industrial context in which this research took place.

## 2.1 Model Driven Engineering

In model-driven engineering (MDE), models are first-class citizens, considered equal to code since their implementation is automated [15]. Models are typically domain-specific, meaning they are constructed using terms and formalisms taken from the domain they are addressing. For example, a networking engineer can design a model of a network by specifying server blocks and the connections between them in the model. These models can then be used for system analysis, combined with other models, or used to generate code. The theoretical benefits of using MDE are therefore many: increased development speed, better software quality, earlier system analysis, more manageable complexity, and interoperability, to name a few [15]. MDE is embraced by many organizations, including the Object Management Group (OMG), IBM, and Microsoft [21].

Domain-specific models are typically created using a *domain-specific language* (DSL). These languages are customized to make it easy to build models for the domain. For instance in the networking example above, there could exist a language construct for a server. Another example is the Unified Markup Language (UML), which is a DSL for software modeling. The constructs available in a DSL are defined by a *metamodel*. A simplified version of the metamodel for UML is shown in Figure 2.1. Models created in that DSL are then said to *conform* to the DSL's metamodel. To differentiate models from their metamodel (which is itself a model), they are often called *concrete* models.

Most operations on models involve transforming the source model(s) into code or other target models. The code that performs this process is called a model transformation. Model transformations are pieces of code (or graphical representations thereof) which describe how to map elements from one set of models onto another. Source and target models could conform to the same or different metamodels.

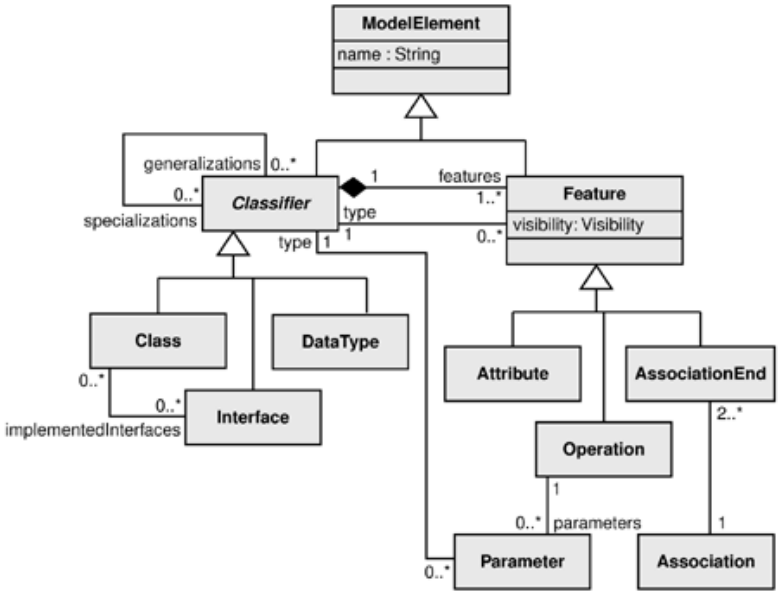


FIGURE 2.1: Simplified metamodel of UML [1]

### 2.1.1 Model Transformation Languages

Model transformations can be implemented using general purpose languages (GPLs), such as Java, or with DSLs designed for the domain of model transformation. The benefit of using a DSL for model transformation is that transformation developers can focus just on implementing the transformation specifics without worrying about additional language syntax required just to get the transformation working. Using GPLs to implement model transformations generally leads to considerably more verbose code than with a transformation DSL [22].

Mens and Van Gorp provide a taxonomy of model transformation [21]. First, transformations differ in what they transform: some transform models into other models while others generate code. Second, depending on the language, transformations can transform either one or multiple source or target models. Third, transformations can be either *endogenous*, where the target models conform to the same metamodel as the source model, or *exogenous*, where the metamodel of the target models differs from the metamodel of the source models. Fourth, transformations can be either *horizontal*, where source and target models correspond to the same level of abstraction, or *vertical*, where the target model is either a refinement or abstraction of the source model. Transformation languages also differ. One distinction is whether they are declarative or imperative. Declarative approaches focus on defining a relation between the source and target models, while imperative approaches focus on the specific steps required to create the target model from the source model. Some languages use both imperative and declarative styles, known as hybrid languages. Another distinction is directionality: whether the transformation can regenerate the source model from the target model (bidirectional) or not (unidirectional). Although not a feature of the language itself, the tooling currently available for development in a given model transformation is also extremely crucial to the usability and success of the language [21].

There currently exist many model transformation DSLs. The Atlas Transformation Language (ATL) is a hybrid language utilizing the the Eclipse Modeling Framework (EMF).

The EMF is a framework that provides capabilities like abstract and concrete syntax development as well as executing transformations. More information on ATL can be found in [23]. Also utilizing the EMF is the Query/View/Transformation (QVT) family of languages. QVT is the OMG standard for MMTs. The QVT family consists of three languages: Core (QVTc), Relations (QVTr), and Operational Mappings (QVTo). Core is a minimal declarative language. Relations, while still declarative, extends Core by adding more user-friendly concepts like object support as well as offering a simple graphical syntax. Both Core and Relations are bidirectional languages. Operational Mappings is imperative, allowing for more complex computations to be written directly into the language, and unidirectional. Each of the QVT languages leverages the Object Constraint Language (OCL) [24] for its query language (e.g. to find a specific set of elements within a model), offering functionality like traversing model elements within a concise syntax.

By focusing on Operational Mappings, we are therefore addressing only a part of the model transformation taxonomy. We do however seek to address all types of QVTo transformations (though per our pragmatist stance our success is still judged within our specific context). Specifically, since QVTo is a model-to-model language supporting arbitrary numbers of source and target models, we consider quality of QVTo transformations with multiple source and/or target models. Since QVTo supports source and target models having different metamodels, we address quality for both endogenous and exogenous QVTo transformations. Similarly, since QVTo can be used to implement both horizontal and vertical transformations, we address quality of both transformation types.

We elaborate on QVTo's syntax in Section 2.2. For more information on the QVT family and its history, see [25] and [15]. For more information on how to choose transformation languages, the reader is referred to [26], [27], and [10], among others. For a more complete list of available transformation languages, the reader is, in fact, referred to wikipedia [28].

## 2.2 QVT Operational Mappings

In this section a selection of the syntax and semantics of the QVTo language are presented. Since the only book dedicated to QVTo is in German [29], we refer to the language overview given by Barendrecht [30]. This section also makes apparent some of the features specific to the model transformation domain that have been incorporated in QVTo.

### 2.2.1 Transformation structure

A *transformation declaration* embodies all transformation code. This declaration first specifies the input(s) and output(s) models, called the *source* and *target* respectively, as well as their corresponding metamodels. The metamodels themselves are defined beforehand using the EMF features and then referenced from within the transformation using a uniform resource identifier (URI). An example of a simple but complete transformation is given in Listing 2.1 utilizing the ABC metamodel from Figure 2.2. Every transformation includes a **main** function which serves as the entry point for the transformation.

```
1 /*
2 *  HelloWorld adapted from
```

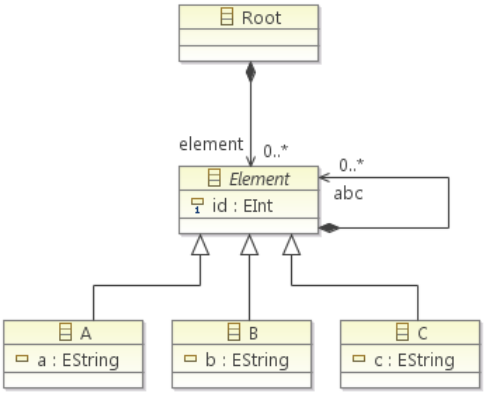


FIGURE 2.2: ABC metamodel

```
3  *  http://www.levysiqueira.com.br/2011/01/eclipse-qvto-hello-world/
4  *
5  *  Transforms a model by converting A-type objects to B-type and
6  *  appending " World!" to a property of the object.
7  */
8  modeltype ABC uses ABC('http:///ABC.ecore');
9
10 // Transformation declaration with one input and one output model,
11 // both of type ABC.
12 transformation HelloWorld(in source:ABC, out target:ABC);
13
14 // Entry point.
15 main() {
16     // Map all root objects of class Root objects with the
17     // Root2Root mapping.
18     source.rootObjects()[Root]->map Root2Root();
19 }
20
21 // Mapping where input and output are of class Root
22 mapping Root :: Root2Root() : Root {
23     // Map A objects to B objects with the A2B mapping
24     // adding them to the target models "element" property.
25     element += self.element->select(
26             a | a.oclIsKindOf(A)    // OCL to only select A-type objects.
27             )->map A2B();
28 }
29
30 // Mapping where input is of class A and output is class B.
31 // when-clause to specify that only objects with id>0 are accepted.
32 mapping A :: A2B() : B
33     when { self.id > 0 } {
34
35     // Assign properties to target object
36     id := self.id;
37     b := self.a + " World!";
38 }
```

LISTING 2.1: A simple QVTo transformation using the ABC metamodel

### 2.2.2 Basic operators and functions

The most important operators are displayed in Table 2.1. All are self-explanatory except the dot accessor, used to execute a function on an object, and the arrow accessor, used on a set to map a function over every element of the set.

| Description | Operator |
|---|---|
| End of expression | ; |
| Assignment | := |
| Equals | = |
| Not equals | <> |
| Dot accessor | . |
| Arrow Accessor | -> |

TABLE 2.1: QVTo operators

The imperative constructs of QVTo, similar to those available in many languages, are `forEach` and `while` loops, variables with block scope, `switch` statements, `return`, and `break` statements. QVTo also allows developers to add `assert` statements for checking conditions and `log` statements to output text to the console.

### 2.2.3 Mappings

Mappings are the core of a QVTo transformation. They specify how an object from an instance of a source metamodel is transformed into an object of a target metamodel instance. Each mapping has a unique signature including the mapping name, input element type, output element type, and optionally additional arguments. Mappings are invoked by using the arrow accessor, as in line 18 of Listing 2.1, or arrow accessors on an object followed by the name of the mapping. Mappings can also include pre- and post-conditions, indicated with the `when` and `where` keywords respectively, such as in line 33 of Listing 2.1. A pre-condition specifies constraints that must be satisfied on the input objects before a mapping can be executed. According to [30] the use case for post-conditions in practice is not yet defined. These conditions are defined using OCL [24].

The body of a mapping consists of three parts: an optional initialization section, a required population section, and an optional end section. The initialization section, specified inside an `init` block, is meant for variable initialization. The population section, requiring no encapsulating block, is meant for populating the output element, i.e. mapping features of the input element to the output element. The end section, specified inside an `end` block, contains code that must be executed after the mapping has occurred. When inside a mapping, the `self` keyword can be used to access the input element, and `result` to access the output element.

### 2.2.4 Queries, helpers, and constructors

A query, defined with the keyword `query`, is an operation to obtain data from the input object(s), for example a subset of its elements from the model. Queries can have additional parameters, but should not have side-effects of their parameters. Helpers, declared

with keyword `helper` (although not described by [30]), are like queries but may have side effects. Finally, constructors can be used to explicitly construct an element from a metamodel using the `constructor` keyword. In this thesis we use the word "function" to refer to mappings, queries, helpers, and constructors collectively.

### 2.2.5 Traces and Resolving

As the transformation executes, a *trace* is recorded for every source element transformed by a mapping. By inspecting these traces one can see for example the order in which mappings were invoked.

Another feature of QVTo is *resolving*. When an object of the source model is transformed to an object of the target model, and then later in the transformation the same source object is referenced, it should not be transformed again. Instead, the reference should be *resolved* to the earlier transformed object. This can be done by calling `resolve` on the source object, which will return the transformed object from the target model. Resolving in the opposite direction is also possible using `invresolve`. To accommodate the case that `resolve` is called but the object has not yet been transformed, the `late` keyword can be used to signify that the object should be resolved instead at the end of the transformation. Internally traces are used to resolve objects.

### 2.2.6 Reuse

QVTo also makes available a number of reuse mechanisms. Transformations can utilize reuse first through *composition*, allowing one transformation to explicitly invoke another transformation using the `transform()` function call. Transformations can also use *extension*, where one transformation can extend another transformation using the `extends` keyword, allowing functions to be overridden. Collections of functions that are reused by multiple transformations can be placed in *libraries*, which start with the keyword *library* rather than *transformation*. Libraries and transformations are both referred to as *modules* in QVTo.

Reuse is also supported at the mapping level. A mapping can *inherit* from another mapping using the `inherits` keyword, causing the other mapping to be called after the `init` section of the first mapping has run. Optionally, the mapping which is being inherited from can be declared abstract using the `abstract` keyword. Mappings can also `merge` a number of other mappings. This causes the specified mappings to be run after the `end` section of the first mapping. Mappings can also use the `disjuncts` keyword, acting like a switch statement to select the appropriate mapping: Upon invoking a mapping which disjuncts other mappings, the first mapping which can accept the input object, typically specified by `when` clauses on the mapping, is executed. If no applicable mapping exists in the disjunct statement for the given input, `null` is returned.

### 2.2.7 Intermediate classes and properties

In addition to the classes and properties already defined within the source and target metamodels, it is also possible to define intermediate classes and properties. Intermediate classes and properties do not appear in the input or output metamodels, but instead are

only used to make the transformation easier to write. They can be used to store transient information during the execution of the transformation, such as when some information is read from the source model early on but then required again later in the transformation.

### 2.2.8 Global parameters and standalone execution

Global parameters can be declared outside of a transformation using the `configuration property` keyword, for instance in the transformation run configuration in Eclipse. They can also be defined when calling the transformation standalone, such as when invoked through Java.

### 2.2.9 Blackboxes

QVTo also allows calling *blackbox* functions written in other languages. Typically blackboxes are used when a function or transformation is too complex to write in just QVTo, and is instead implemented in a language like Java. One can also encapsulate for instance QVT Relations functions inside a blackbox so it can be invoked from QVTo. Blackbox transformations are defined using an empty transformation signature.

### 2.2.10 Environment

Three implementations of the QVTo specification are available: The Eclipse QVTo project [31], SmartQVT [32], and Borland Together Architect [33]. The Eclipse QVTo project and SmartQVT both leverage Eclipse Modeling Framework (EMF) plugins [34], which are provided as defaults when the Eclipse bundle for modeling engineers. The Eclipse QVTo project, which we also refer to as the Eclipse QVTo implementation, adds a QVTo launch configuration, where developers can specify which transformation to run as well as the desired input models from within Eclipse. This implementation also provides some facilities for debugging, for example setting breakpoints in a transformation allowing one to inspect variables and step further into the code.

## 2.3 Industrial Context

This research has been carried out with ASML N.V. [14]. Specifically, we worked together with the modeling group of ASML's Control Architecture Reference Model team, called the CARM2G team, referred to here typically as just the CARM team. This team maintains a set of DSLs to describe a part of ASML's machine architecture containing the process controllers, including software as well as the hardware on which they are executed. These models are then used to perform activities ranging from real-time schedule analysis to servo controller initialization during machine startup [13]. To support this, more than 130 QVTo modules have been developed by the team, totaling almost 20,000 lines of QVTo code. Most transformations maintained by the team are exogenous transformations combining models at the same level of abstraction or performing refinements.

The CARM team uses the Eclipse QVTo implementation, described in Section 2.2. However, rather than using the QVTo launch configuration, their transformations are always

run in a standalone context (i.e. invoked via Java) since the transformations must also run on the machine itself, where no Eclipse installation is available. Also because transformations are executed directly on the machine during startup, code performance is extremely important. Tests are written for the transformations using JUnit, SVN is used for versioning the team's code, and while a bug tracker exists for the transformations, it is not heavily used by the team internally.

For GPLs (specifically, C, C++ and Python), ASML has extensive support for quality control, including strict coding standards and tools. For the past 1.5 years, ASML has used the TIOBE TICS [35] tool, which measures code metrics such as code test coverage, cyclomatic complexity, adherence to coding standards, code duplication, dependencies, and dead code in GPLs. These metrics are measured at least once a week on the code base. Coupled to the TICS tool at ASML is a developer dashboard, where component owners can specify metric goals for their components and track their own progress towards those goals. According to the quality assurance department [36], developers consider this quality tooling very useful. Code quality control is therefore already a normal part of software development at ASML. Part of the motivation for this research is to move closer to being able to provide such quality support and focus to QVTo development.

The CARM team is comprised of six developers from diverse backgrounds. Two of the developers have doctoral degrees in computer science, one related to model transformations and one in formal methods. The rest of the developers have higher-education degrees in computer science, except one which has a higher degree in mechanical engineering. These developers work with QVTo on at least a weekly basis.

# Part I

# Formalizing QVTo Quality

# Chapter 3

# Quality

The field of software quality is vast. Here we provide a brief overview of the work most relevant to model transformation quality which served as a starting point for this research. Additional work is presented with the formal literature review in Chapter 5.

Notably, we do not adopt a definition for quality *a priori*. However, we do note at times throughout this chapter which perspectives and assumptions we are adopting from the start in order to most effectively answer our research questions, such as with the pragmatist stance is Section 1.1.

First we present some of the perspectives one can assume when assessing quality in Section 3.1. Then we present the existing quality models most relevant to model transformations in Section 3.2. Finally we discuss quality model frameworks in Section 3.3, which can be used to develop a new quality model.

## 3.1 Perspectives

There are a number of lenses one can use with which to look at quality. Whereas the philosophical stances presented in Section 1.1 help direct *how* to carry out research, these perspectives influence the *content* a definition of quality will have. In order to design a quality model and to understand the value of existing software quality models, it is essential to understand which perspectives are adopted by the researchers. Here we given an overview of these perspectives. In Section 3.2, we relate the models discussed back to these views.

### 3.1.1 Internal versus external quality

Ferenc, Hegedűs and Gyimóthy [37] provide a detailed discussion of software quality. Although their discussion is primarily in the context of software evolution, many of their insights are applicable to software quality in general.

They identify one of the most basic features of a quality definition to be whether it considers *internal quality* or *external quality*. Internal quality is measured by looking at properties *inside* the software product, for example by analyzing its source code. In

addition to code metrics, diagrams of software architecture can also be part of an internal quality model. External quality, on the other hand, is measured on the product in use, therefore typically while the software is executing. Measuring bugs found by end users is an example of an external quality measurement. The ISO/IEC 9126 [38] standard on software quality also makes this distinction, distinguishing internal software metrics which do not rely on the software executing from metrics which are only applicable to running software. ISO/IEC 9126 and its successor are described in more detail in Section 3.2.1.

In this thesis we focus on internal quality. We feel that a focus on internal quality has the highest potential to provide fine-grained insights that are immediately actionable in practice, and is therefore the most useful to help developers assess and improve their own code. Moreover, it is often assumed in software metrics research that higher internal quality results in higher external quality [39]. Because of our focus on internal quality, we also often use the terms *code quality* and *transformation quality* interchangeably.

### 3.1.2 Five views of software quality

Kitchenham and Pfleeger [39] further elaborate on five views of software. They classify definitions of quality, looking at both software products and processes. In particular these views can drive what design decisions and processes are made while developing a product.

- The **transcendental view** considers quality as something that can be recognized but not defined. An example of this view in software is considering a product high-quality when it "delights users" [39].
- The **user view** considers quality as fitness for purpose. Those who adopt this view consider software to be high-quality when its functionality precisely matches the user's needs given the specific context in which it is used. Software usability is most closely related to this view.
- The **manufacturing view** considers quality as conformance to specification, focusing on processes to create a product correctly. Adopting this view for software suggests that conformance to process results in high-quality software products.
- The **product view** considers quality as tied to inherent characteristics of the product. Unlike the user and manufacturing view, this view is most closely related to internal quality rather than external. In software this view means that overall software quality can be assessed by measuring internal properties. However, the authors caution that more research is required to confirm the causal link that high internal quality actually loads to high external quality (i.e. quality in of software during use).
- The **value-based view** considers quality as dependent on the amount a customer is willing to pay for it. Here, software quality is equated with market value. This view in particular has implications for design decisions, since trade-offs will be made between cost and elements from the other quality views.

Although our pragmatist stance could be used in conjunction with any of these views, our focus on internal quality suggests that our definition of quality will be most closely related to the product view.

### 3.1.3 Direct versus indirect measurement of quality

With respect to model transformation quality, van Amstel [40] distinguishes between *direct* versus *indirect* measurements of transformation quality. Direct measurements of quality measure the transformation itself. In indirect measurements, other MDE artifacts are measured instead to assess the transformation quality, such as metamodels involved or concrete output models produced by the transformation. Van Amstel notes that, although not impossible, it can be difficult to assess internal quality using indirect measurements. Therefore, we prefer direct quality measurements in this research.

## 3.2 Quality Models

A *quality model* can be defined as "the totality of features and characteristics of a conceptual model that bear on its ability to satisfy stated or implied needs" [38]. Software quality models typically consist of a collection of high-level *quality goals* which can each be broken down into lower-level *quality attributes* of the software. The nature of the goals and attributes represents the chosen perspectives of quality. By analyzing (e.g. measuring) these attributes, one can draw conclusions about how well the software achieves the quality goals. Although there are numerous software quality models in research, we present only two here.

### 3.2.1 The ISO Standard

ISO/IEC 9126, the international standard for the evaluation of software quality, was issued in 1991 to provide an explicit definition of software quality goals [38]. In 2011 it was succeeded by ISO/IEC 25010 [41], in which two quality models are defined, each representing a different software quality view. The first is a *quality in use* model, which represents the user view. In the quality in use model, five quality goals (there called *characteristics*) are defined which are subdivided into quality attributes (there called *subcharacteristics*). These goals and attributes relate to the outcome of interactions when a product is used in its particular context of use, such as satisfaction, safety, and usability.

The second model contributed by ISO/IEC 25010 is a *product quality model* representing the product view. The product quality model defines eight quality goals with attributes. This model relates to static properties of software as well as dynamic properties of the computer system. The quality goals defined by the software product quality model are described below:

- **Functional suitability**: The degree to which the product provides functions that meet stated and implied needs when the product is used under specified conditions.

- **Reliability**: The degree to which a system or component performs specified functions under specified conditions for a specified period of time.
- **Operability**: The degree to which the product has attributes that enable it to be understood, learned, used and attractive to the user, when used under specified conditions.
- **Performance efficiency**: The performance relative to the amount of resources used under stated conditions.
- **Security**: The degree of protection of information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them.
- **Compatibility**: The degree to which two or more systems or components can exchange information and/or perform their required functions while sharing the same hardware or software environment.
- **Maintainability**: The degree of effectiveness and efficiency with which the product can be modified.
- **Transferability**: The degree to which a system or component can be effectively and efficiently transferred from one hardware, software or other operational or usage environment to another.

These quality goals therefore represent a consensus on what, in a most general context, represents product quality in software. However, because it still may not represent the most *useful* quality model for QVTo transformations, which is what we seek with the pragmatist stance, we still do not adopt these quality goals (nor the quality model attributes) from the standard *a priori* in this research.

### 3.2.2 A quality model for transformations

Van Amstel [42] proposed a quality model for model transformations with respect to development and maintenance. He defined the following quality goals: understandability, modifiability, reusability, modularity, completeness, consistency, and conciseness. Most are self-explanatory. Consistency includes attributes like maintaining a uniform programming style. Completeness includes for instance how well the software fulfills its specification. Completeness is the only one which considers some aspects of external quality in addition to internal.

For these quality goals, van Amstel defined sets of model transformation metrics which could be measured in order to assess how well each goal was fulfilled. In particular metrics were defined for the model transformation languages ATL and ASF+SDF, though most can be applied to QVTo as well. In total 66 metrics were identified. The metrics could be categorized as size metrics (e.g. number of rules), complexity metrics (e.g. number parameters per rule), modularity metrics (e.g. number of imported units), inheritance metrics (e.g. number of abstract rules), dependency metrics (e.g. number calls to rules), consistency metrics (e.g. number unused rules), input/output metrics (e.g. number input models), and language-specific metrics (e.g. rules with input patterns in ATL).

The metrics were based in part on the smaller set of quality metrics proposed by van Amstel and Nguyen for ATL, ASF+SDF, and QVTo [6].

Although tempting, we also do not adopt this model *a priori*. This is firstly because a model constructed for ATL or ASF+SDF may not be directly applicable to QVTo. At the very least, this would prevent any QVTo-specific metrics from being included in the quality model. Secondly, the model presented by [42] has not been validated well-enough to be assured of its usefulness in practice in a general context (the validation approach used by van Amstel is discussed in more detail in Chapter 7). Nonetheless, van Amstel's work serves a strong basis for our research in QVTo transformation quality.

### 3.2.3 Why metrics?

Although we attempt to approach our research questions without a predisposition toward metrics, it is a constantly recurring theme in code quality research, resulting in it being a difficult to separate ourselves from the lurking assumption that the quality model we develop will rely extensively on metrics. *If all you have is a hammer, then everything looks like a nail.*[1] We acknowledge this bias, and use this section to expound on some of the benefits of using concrete metrics in quality models.

Numerical assessments of product quality can be very valuable to organizations and often used in decisions such as where to allocate more resources [37]. This is because qualitative data is hard to interpret, always subjective, and time-consuming to collect. In fact, metric programs are required to be integrated into the software development process to reach higher Capability Maturity Model levels [44]. Likely for these reasons (although not always made explicit), quality models developed in research are often assumed to use metrics as their underlying implementations.

That said, there are a number of shortcomings of using metrics. Firstly, the effectiveness of using metrics is still highly controversial [37]. In a review of 15 different predictive models for software maintainability, for example, Riaz, Mendes, and Tempero [45] found little evidence of metrics being effective at all.

Alternatives to metrics include for example identifying design patterns, like those that have already been extensively catalogued for many general purpose languages [5]. According to Syriani and Gray [16], the lack of documented design patterns is in fact one of the primary challenges in assessing model transformation quality.

Moreover, it is important to realize that metrics need not be used as a strict indicator for high or low quality. Metrics also serve as the underlying mechanism for many visualizations considered useful to understand quality. It can also be informative to look at changes in metrics over time or across a code base. Therefore, in this research, we use the term metrics broadly, encompassing any measurements of software which can in turn be used to gain insight into its quality.

[1] A caution to me from Alexander Serebrenik, my advisor, originally attributed to [43].

1. Develop a set of corporate or project goals.
2. Generate questions that define those goals as completely as possible in a quantifiable way.
3. Specify the measures needed to answer those questions and track conformance to the goals.
4. Develop mechanisms for data collection
5. Collect, validate, and analyze data to provide feedback for corrective action.

FIGURE 3.1: The steps of the GQM paradigm [2]

## 3.3 Quality Model Frameworks

Quality models can be constructed systematically by following a *quality model framework*.[2] In general, quality model frameworks have the following principles [46] based on ISO/IEC 9126:

- Decompose model quality into a hierarchy of characteristics.
- Use clear-single-word labels for each characteristic and define them with a single, concise sentence.
- Define metrics for each characteristic.
- Provide detailed procedures for conducting metric evaluations.

We describe two specific quality model frameworks next.

### 3.3.1 Goal/Question/Metric

The Goal/Question/Metric (GQM) paradigm was introduced by Basili [2] to help define software measurements. Specifically, it helps researchers choose and interpret software metrics by providing a framework within which one can create goals for the software that are tailored to the research context. Applying the GQM approach involves five steps:

A relatively general approach, GQM has been used by a number of researchers in empirical software engineering and model-driven engineering (e.g. [27, 46]).

[2] A quality model framework may also be referred as model quality framework or model quality model [46], and we could probably even get away with quality model quality model since we are specifically talking about quality models, but we will stick with quality model framework.

1. Identify quality goals (e.g. maintainability, increased productivity).
2. Identify target objects that impact the quality goals (e.g. model transformations, metamodels, tooling).
3. Identify the quality-carrying properties of the target objects.
4. Specify how to evaluate the quality-carrying properties, either with metrics or subjective evaluation.
5. Specify the association links between the quality-carrying properties and quality goals.
6. Review and evaluate the framework in practice. This evaluation considers characteristics such as completeness, flexibility, and possibility to be adopted.
7. Execute by implementing the quality-carrying properties and evaluation.

FIGURE 3.2: The steps of the MDE quality framework [3]

### 3.3.2 Quality Framework for Model-Driven Engineering

Mohagheghi and Dehlen [3] present a 7-step quality framework specifically for model-driven engineering (MDE) artifacts. We refer to this as the *MDE quality framework*. A depiction of applying the framework is given in Figure 3.3 and we elaborate on the seven steps in Figure 3.2. *Quality-carrying properties* correspond to quality attributes in our terminology.

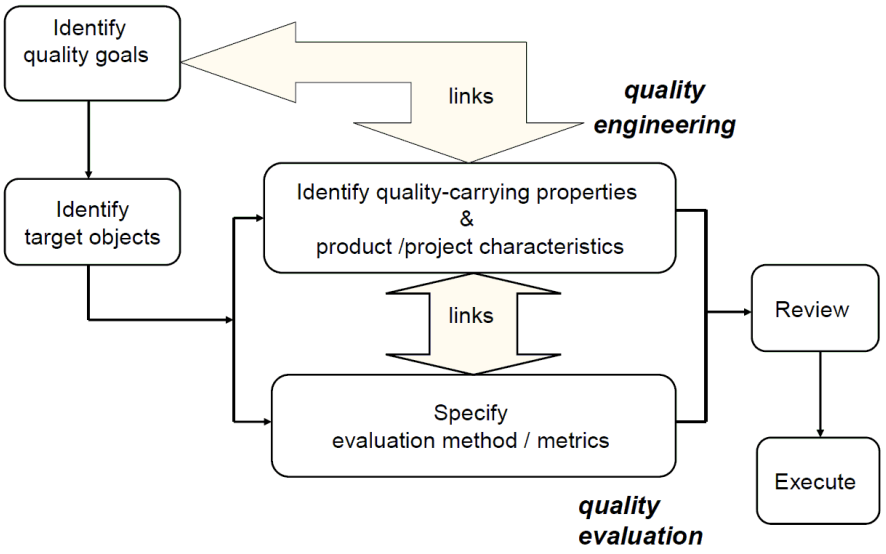


FIGURE 3.3: Applying the MDE quality framework [3]

The authors stress the importance of involving stakeholders when defining quality goals and attributes. They also stress that traceability links between the quality-carrying properties and the rationale for why they are important should be maintained. Finally, the authors state that in order to lead to successful results, there must be adequate tool support for the evaluation steps.

As an illustration, an example (partial) model resulting from this framework is presented in Figure 3.4. The MDE quality framework has not yet been applied extensively in practice, to our knowledge only having been partially applied once in research ([47], though Dromey's product quality framework [48], on which the MDE framework is based, has been used more often, e.g. [49]).

| Target Object | Quality Goal | Quality-carrying Property | Evaluation |
|---|---|---|---|
| **Transformation process** | *High performance* | Effective transformation engine | Measure performance |
| | | Select appropriate transformation approach | Measure performance |
| **Transformation model / rules** | *Preservation of consistency* | Enforce consistency by tools | Consistency analysis tool, measuring consistency before and after transformation |
| | *Reusability* | Modularization, i.e. specialize and chain transformations, and rule inheritance | Inspection |
| | *Simplicity* | Few number of rules, i.e. modularization | Measure complexity in the number or size of rules |
| | | Appropriate algorithm | Measure the complexity of algorithms |
| | | Simple output models | Measure complexity and size of the output model |

FIGURE 3.4: An example partial model produced with the MDE quality framework [3]

We have chosen in this research to follow the MDE quality framework to develop a quality model for QVTo because it is specifically designed for MDE. Namely, the differentiation the authors make between target entities such as *transformation* versus *transformation process* is an intuitive way to capture different angles that all affect quality inside a single quality model. GQM, on the other hand can lead to unintuitive collections of attributes. For example, Kolahdouz-Rahimi et al. [27] used GQM in developing an evaluation framework for model transformation approaches. There, GQM led to quality attributes such as "interoperable with Eclipse", "development effort", and "complexity" all under the same heading of "Functionality", even though each attribute relies on very different units of analysis. We therefore use this research as an opportunity to investigate whether the MDE framework can lead to a cleaner separation of concerns within the quality model.

### 3.3.3 Applying quality model frameworks

Quality model frameworks can then be applied using a number of approaches discussed by Moody [46]. These approaches are closely coupled with the philosophical stances presented in Section 1.1. Although they are presented in [46] as alternatives to the GQM or MDE quality frameworks, we believe they can be used as an approach to apply either framework. We enumerate the approaches most relevant to this research:

- **Theory-based** approaches employ deductive reasoning. Here a quality model framework leverages only theory, for instance by applying theory taken from other disciplines. This is most closely related to the positivism stance. Although this approach is the most commonly taken in developing quality models in software quality research, the author of [46] concedes that there are no *a priori* reasons why this is likely to be effective in practice.
- **Observation-based** approaches are inductive, advocating a bottom-up manner to building a quality model. The advantage of using such an approach is that the resulting model will, by construction, already have a high level of empirical validity, provided that a suitable number of observations are made. To the knowledge of the author of [46], this approach has not been used in quality model research, but is common in many other disciplines. This is most closely related to the constructivism philosophical stance.
- **Synthesis** approaches employ analytical methods where existing proposals are combined into a unified model. This approach has been used in some research in information systems with success, but has not been applied to quality research in a rigorous manner. These approaches could be considered closest to the pragmatism philosophical stance.

Although similar, these approaches to developing quality models are not equivalent to the notion of philosophical stance presented in Section 1.1. Most importantly, philosophical stances also direct how to evaluate the success of a theory, whereas these approaches are just techniques. These approaches are also not identical to the research techniques also mentioned in Section 1.1 (e.g. action research, mixed-methods research), since these approaches have been tailored to the process of building quality models. The synthesis approach for example is an application of mixed-methods research to build a quality model.

We elaborate on our approach for building a quality model for QVTo, including choosing one of the approaches above, in Chapter 4.

# Chapter 4

# Approach

Our first research question asks to formalize quality assessment for QVTo transformations. As in related work from Chapter 3, we have chosen to do this by developing a quality model for QVTo transformations.

As discussed in Section 3.3.2, we follow the MDE framework to construct the quality model. Arguably more importantly, in following the framework, we have chosen to use a *synthesis* approach. As explained in Section 3.3.3, synthesis approaches combine top-down methods (e.g. theory) with bottom-up methods (e.g. observations taken from the field), fitting well with the pragmatist stance.

Specifically, we seek to leverage existing academic research in software quality, but we do not assume that quality models there are immediately applicable to QVTo. Avoiding this assumption is required by the pragmatist stance, since the most useful quality model for QVTo will heavily depend on its current usage in practice and factors such as the maturity of language implementations and tools. In fact, in Rahim and Whittle's survey of model transformation verification and assessment practices [50], the authors stress that current research is in need of more empirical studies: "Connections must be grounded in real-world experience and, to date, they have been made either only on intuition or based on rather limited empirical studies." Therefore we combine using existing research with practical observation. Moreover, since according to the pragmatist stance the success of the quality model is also judged in a context-dependent way, it is more likely for the quality model to be successful if it leverages observations from the field when constructing the model as well.

We organize our synthesis approach as a broad *exploratory study*. In the remainder of this chapter we describe the details of the approach. How we apply the MDE framework is described in Section 4.1. Since the exploratory study makes heavy use of empirical and qualitative methods, we provide an introduction of empirical methods in Section 4.2. Then we describe the exploratory study in detail in Section 4.3.

## 4.1 Applying the MDE Framework

The MDE framework was introduced in Section 3.3.2. Below we elaborate on how we carry out each of the seven steps of the framework to build a QVTo quality model using a synthesis approach.

### 1) Identify quality goals, 2) Identify target objects, 3) Identify quality attributes, and 5) Associate quality goals with quality attributes

Since we seek to build much of our notions of QVTo quality through a synthesis of methods, a number of steps from the MDE framework can be carried out simultaneously. Specifically, to identify the quality goals, target objects, and quality attributes that are most relevant for QVTo, as well as the links between them, we conduct a large *exploratory study*. This exploratory study therefore lies at the crux of building our quality model. We describe the exploratory study approach in detail in Section 4.3.

### 4) Specify how to evaluate quality attributes

We then specify the evaluation strategies for each quality attributes identified in the previous steps. Again leveraging a synthesis approach, we do this through a combination of information gathered throughout the exploratory study and our own knowledge of how certain quality attributes could be measured in QVTo. These evaluation strategies are identified while constructing the quality model, described in Chapter 6.

### 6) Evaluate the quality model

The quality model for QVTo must then be evaluated. Specifically, we focus on validating the relationships between the quality goals and quality attributes. An extensive component of this research, our quality model evaluation is presented in Chapter 7. We also evaluate how well our model agrees with the existing consensus on software quality by checking the conformance of our QVTo quality model to the software product quality model defined by ISO/IEC 25010 (Section 6.3.1).

### 7) Execute

As described in Section 3.3.2, the quality model is *executed* by implementing the evaluation procedures inside a tool. Rather than directly implementing one large metrics-measuring tool, however, we maintain the pragmatist stance and instead leverage the data from the exploratory study and evaluation to try to identify what tools may be most effective and useful based on our quality model. It is this analysis and the subsequent implementation of a tool that is the subject of Part II of this thesis.

## 4.2 Empirical and Qualitative Methods

The exploratory study relies heavily on empirical methods. Empirical research is research which gains knowledge through observing or experiencing phenomena. In particular we make use of *qualitative* methods. Qualitative data are data represented by words or pictures as opposed to numbers, as is used in quantitative research. These methods were designed originally by social science researchers to study the complexities of human behavior. Qualitative methods are used since they lead to very rich, descriptive information. Moreover, qualitative research can often collect data about phenomenon

that *cannot* be obtained using quantitative measures [51]. For this reason, qualitative methods are often recommended for exploratory research [17]. However, this richness also makes the results of qualitative research often more labor-intensive to analyze than quantitative data [52].

In software engineering, qualitative methods are often considered "softer" or "fuzzier" than other empirical methods since they are more difficult to objectively summarize. However, to understand how software is developed (and in turn how it comes to have high or low internal quality), one must understand how individual software engineers develop, as well as how teams and organizations coordinate their efforts [17]. Therefore, it in fact lends itself to qualitative methods [52]. Moreover, despite their fuzzy or even imprecise connotation, qualitative methods have been used extensively in social science fields for a very long time, so there exists an abundance of material on how to conduct these methods effectively.

Each qualitative method has its own strengths and weaknesses. Therefore its often necessary to combine qualitative methods in order to fully understand a problem. Combining methods (a.k.a. using mixed-methods) to obtain a more acute picture of reality is called *triangulation* [52].

## 4.3 Exploratory Study

In this section we describe the approach to our exploratory study. As described in Section 4.1, the primary goal of the exploratory study is to determine the quality goals and quality attributes most relevant for QVTo using a synthesis approach. As stated in Section 4.2, we maintain a focus on *qualitative data* because it offers some of the richest insight into how developers write code. The alternative here would be gathering quantitative data, for example measuring metrics on a code base and comparing them to some variable like the number of bugs. However, this is not feasible because there are not reliable sources of information like bug data.

To combat the pitfalls of individual qualitative methods, we use per the recommendation in Section 4.2 the triangulation of three methods. First, we conduct interviews of QVTo experts. Second, we perform a broad review of existing material including both literature and other sources. Third, we use introspection, wherein the researchers themselves learned QVTo and reflect on quality issues encountered. By observing where data from the methods overlap, we get a rich, generalizable view of what is most relevant for QVTo quality. Furthermore, gathering the quality goals and attributes in this exploratory manner already provides high empirical validity, as mentioned for the observation-based approach discussed in Section 3.3.3. Since model validation can be a daunting task, this validity by construction is extremely valuable.

### 4.3.1 Expert Interviews

The first component of our triangulation approach is expert interviews. A specific focus on developers was chosen because then we can leverage potentially years of best practices and insight involved with becoming an experienced developer. Furthermore, interviews can often draw out richer information than methods like questionnaires since the interviewer can immediately follow up on answers.

In our approach, we perform *semi-structured interviews* of QVTo experts, where mixture of specific and open-ended questions are used, allowing the interviewer to gather both specific and unanticipated information. The primary selection criteria for interview subjects were that they are comfortable and knowledgable about QVTo. Since we had access to the team, developers of the CARM team were selected. The biggest drawback to using interviews is that an interviewer's questions and manner always introduce bias. Semi-structured interviews also have the downside that, due to their open-ended nature, responses are not always comparable between interviewees [17].

Seaman [52] provides a number of additional guidelines for performing interviews in software research. Interviews should begin with a explanation of the research being conducted, but contain only a minimal amount of information as to minimally bias the interviewees. Although the interviewer should come prepared with specific topics, it is in general best to allow the interviewee to ramble, since that results in richer data. It should therefore be made clear that there are no "right" answers. An *interview guide* can be used by the interviewer to organize the interview, typically consisting of a list of open-ended questions. Interview guides should not be shown to the interviewee, since it again could bias their answers. Since it is difficult to take notes and conduct an interview at the same time, it is best for interviews to be audiotapes and then later be transcribed verbatim or made into summaries called *field notes*. Writing field notes can take up to three hours per hour of interview audio, whereas transcribing can take up to eight hours per hour of audio [51].

Hove and Anda [51] also share their experiences from conducting 280 semi-structured interviews in software engineering research. First they stress that it is important for researchers utilizing interviews in their research to describe in detail how interviews are conducted, making the areas where bias could be introduced as apparent as possible. This information includes the duration and location of the interviews, as well as the contents of any interview guides used. Although feigning ignorance is sometimes used as a tactic in social sciences to help interviewees be more verbose, they stress that in the field of software engineering, it is important for the interviewer to have extensive knowledge of the interview topic in order to gain legitimacy with the interviewee. In general "yes"/"no" as well as "why" questions should be avoided, since they elicit less verbose responses. Instead, ask more "what" and "how" questions. Questions that describe how interviewees work as well as opinion questions often work well to extract richer information. Finally, questions with too many details can be hard to answer, since interviewees will not often remember.

All of the above recommendations were taken into account for our interviews. In particular, a one-page interview guide with open-ended questions was prepared, presented in Figure 4.1. Furthermore, although measurable attributes are an important component of quality models, a deliberate attempt was made to not focus exclusively on metrics during the interviews. Instead, the focus was placed on more abstract quality concerns. This way we do not restrict ourselves to our own or the developer's current expectations for what can be easily measured in code, which could result in many important concerns that are not directly measurable not being mentioned. Should such unmeasurable concerns arise, the researchers can reflect on the most effective way to capture them in a tool or metric later on. To draw out all relevant information, interview guide questions were formulated in four categories: general development process, quality, other languages, QVTo, and metrics and visualizations. Within those categories, questions about typical

development practices, common refactorings, tooling, experiences learning QVTo, and frequent frustrations were asked. All interviews were audio recorded.

To test the question clarity and interview length, a mock interview was performed beforehand with an independent developer with similar background, specifically an Master computer science student who also worked part-time as a C# developer. This mock interview, also tape-recorded, also served to make the interviewer (the first author) more aware of her own habits that may influence the interviewee's answers. From this mock interview, many observations were gained. First, the interviewer did not elaborate on answers as much as expected, therefore more leading questions (e.g. "anything else?") should be asked. All questions were also covered in a shorter amount of time than the expected 1 hour, therefore the expert interview length was shortened to 45 minutes. The interviewer had the tendency of giving many specific examples along the question, which strongly biased the interviewee's answers. Regarding question clarity, the interviewee felt that many questions came "out of the blue", revealing the delicate balance between open-endedness and providing enough context. Therefore, it was decided that more broad questions on quality should be asked first to get the developer more comfortable with the general line of questioning. Also, questions on best practices seemed difficult to answer for the interviewee, but when pushed yielded interesting data, such as personal pet peeves. For example, the developer is very annoyed by others committing code without unit tests, and wishes there were some automated check that test code is submitted with all new functionality. The interviewee was however hesitant in answering questions about other developers, and instead it was more effective to ask about struggles the developer personally had while learning. Some words also misled the interviewee. For example, the word "metrics" and "design patterns" had a very narrow interpretations, suggesting that it's best to avoid them. Finally, it also proved interesting to ask customer-facing questions, for example what kinds of bugs are typically encountered by end-users, since these provide an additional perspective. All of these observations were taken into account in creating the final interview guide already presented in Figure 4.1.

A pre-interview questionnaire was first distributed to all participants to gather background information. This questionnaire is presented in Appendix A. A brief meeting with the participants was also arranged to explain the overall project and allow time for questions. To guarantee that the interviewer was knowledgable on the interview subject, the expert interviews were conducted only after the introspection component described in Section 4.3.3.

### 4.3.2 Existing material review

To compensate for the drawbacks of using the observation-based approaches of interviews and introspection, a review of external materials was performed to identify existing information on QVTo quality. This review itself was composed of two subcomponents: a systematic literature review of existing research, and a review of informal sources. This component, and in particular the systematic literature review, serves as the theory-based component of our synthesis approach. The review of informal sources, however, is also essential since the current amount of scientific literature on QVTo is limited, and, like with expert interviews, much information can instead be gained by less formal means.

A *systematic literature review* is a means of identifying, evaluating, and interpreting all available research relevant to a particular topic area [53]. Per the guidelines for

# QVTo Expert Interview Guide

**Logistical info** Name, date, location, start/end time

**General development process**

- Can you describe a typical development process for you?
- What are the key triggers for development? (bug fixing, etc)
- How do you use SCM or bug tracking during a project?
- What other tools do you utilize?
- How often do you personally refactor code?

**Quality**

- If you looked at a piece of QVTo code that you considered high quality, what traits would it have? And low quality?
- What problems do you or your users typically encounter in the code?
- How do you typically discover a problem?

**Other languages**

- What are the biggest differences between developing in QVTo and other languages?
- What do you like/dislike about QVTo?
- Are there tools/IDEs you have had for GPLs or other tools you would like?
- Do you use design patterns in QVTo or other languages?

**QVTo**

- What parts of QVTo development did you or others struggle with most?
- What kind of problems do you typically encounter while developing?
- For your team, what parts go more smoothly than they did in the past?
- Can you identify specific practices for QVTo you try to use in your code?

**Metrics and visualizations**

- Have you used tools which measure quality, for example TIOBE TICS?
- Have you used visualizations? Which?

FIGURE 4.1: Interview guide used by interviewer during QVTo expert interviews

| Topic | Keywords | Included technologies |
|---|---|---|
| Best practices | best practices, standards, guidelines, recommendations | QVTo, model transformations, ATL, QVTr |
| Metrics | code metrics, quality metrics | QVTo, model transformations, ATL, QVTr, Java |
| Tools | tools, tooling, developer tools, productivity, plugins | QVTo, model transformations, ATL, QVTr |
| Limitations | limitations, difficulties, problems, drawbacks, challenges | QVTo, model transformations |
| Quality | code quality, assessment, verification | QVTo, model transformations, ATL, QVTr, Java |

TABLE 4.1: Topics, keywords, and technologies used as inclusion criteria during the systematic literature review

conducting systematic literature reviews given by Kitchenham [53], a review protocol was first developed. This protocol reduces researcher bias in performing the review. Among other information, the protocol specifies a number of topic areas to be included in the review. These topics were chosen largely based on our pragmatist stance, since any of those subjects could affect how quality presents itself in practice. We present the topics along with specific search terms used in Table 4.1. Tooling is included as a topic because of its importance to writing high-quality code.

For topics where the amount of research available specifically for QVTo would still be quite limited, the *inclusion criteria* dictating which papers to include in the review were set to include a broader range of technologies, such as other transformation languages or code in general. Notably, we at times explicitly searched for ATL, which has imperative constructs like QVTo, and QVTr, which comes from the same language family, since they have many similarities with QVTo. The technology criteria used for inclusion for each topic are also presented in Table 4.1. No papers were excluded according to publication date.

Google Scholar was used to perform all searches, therefore covering a large portion of peer-reviewed online journals as well as books and other non-reviewed journals [54]. Offline and non-English language sources were not reviewed. When there were many results when searching by the keywords in combination with any technology other than QVTo, only the first two pages were reviewed. The title, publication date, and a list of extracted key points from the downloaded papers were recorded in a document file organized by topic (and when a paper appeared in two topics, it was recorded under the first). Whether a paper provided explicit evaluation of their contributions was also marked in the document to show increased credibility. Once the review preparation was complete, the literature review was conducted by a single researcher and lasted approximately two weeks.

The second subcomponent of the existing material review is a review of informal, non-scientific sources. Specifically, these sources included forums, blogs, tutorials, online contests, bug trackers, and other websites. These informal sources were found first by searching using the same keywords and technology terms as used for the literature review, presented in Table 4.1, but in Google rather than Google Scholar. From the sources found, all information that might be relevant to QVTo quality assessment or improvement was recorded along with a traceability link to the original source. Forums were reviewed because they can expose first the most common problems end users have with the language, as well as the recommendations that more experienced developers

provide. Similarly, bug trackers can expose where users are still having trouble in the language implementations. Blogs are reviewed since they may mention opinions or other information about what it's like to program in QVTo. Website tutorials, in addition to some potentially interesting commentary within the tutorial itself, can also have revealing comments added by other users. Finally, online contests are mentioned here explicitly because, as part of efforts to increase the prevalence of MDE, there exist some contests run online related to model transformations. A number of ASML materials related to code quality were also reviewed, for example materials from a lecture on code quality given by the CEO of TIOBE, the creator of the TICS tool [35]. The results of our existing material review are presented together with the findings of the rest of the exploratory phase in Chapter 5.

### 4.3.3 Introspection

The third component of the exploratory study was introspection, namely learning QVTo by the first author of this thesis. Here, a series of QVTo tutorials and examples was followed (e.g. [15], [55], [30]) over the course of approximately two weeks. During the learning process all aspects related to quality were recorded, in particular difficulties and realizations about better ways to implement certain functionalities.

Such introspective analysis is useful because it gives first-hand experience with the subject matter and gives much insight into what it's like for people just learning QVTo. Since at least in the CARM team the QVTo programmers started in other languages and also still frequently program in other languages, beginner habits can provide insight into some common problems that people may have. Furthermore, it allows the expert interviews to be conducted more effectively, as mentioned earlier. The Eclipse QVTo implementation was used for the entire learning process. This was also the implementation instructed to download in each tutorial, suggesting that this is the most prevalently-used QVTo implementation.

# Chapter 5

# Exploratory Study Findings

In this chapter we present the findings from each of the three components of the exploratory study, following the approach described in Chapter 4. Observations and some reflection on techniques used in each component are also discussed. Best practices and difficulties gathered from the expert interviews, one of the secondary contributions of this research, are presented in Sections 5.1.2 and 5.1.3, respectively.

## 5.1 Expert Interviews

Four of the six QVTo developers of the CARM team were interviewed by the first author per the approach described in Section 4.3.1. Only four developers were interviewed because of the time constraints of the team and because four seemed sufficient to obtain generalizable information within the context of their team. The four selected were considered by the team to be the most experienced in QVTo. Interviews were all conducted on the same day. The benefits of audio-recording were clearly evident, guaranteeing that no information was missed due to multitasking.

The allocated time of 45 minutes proved an appropriate amount of time for each interview, with only one requiring 50 minutes. Therefore the mock interview provided a useful indication of the required time. The conversation between the interviewer and interviewee generally flowed more smoothly than in the mock interview. We suspect this was the case first because the mock interview interviewee was aware that it was a mock interview, which makes the conversation less natural; and second because the developers interviewed work with the subject matter on a more frequent basis than the mock interviewee. Nonetheless, the interviewing habits noticed by the interviewer during the mock interview (described in Section 4.3.1) seemed to greatly increase interview effectiveness. It was also clearly beneficial that the interviewer had studied QVTo beforehand, supporting the claim presented in Section 4.3.1.

Overall interviews were less repetitive than expected, where the preferences of each interviewee were clear. For example, two interviewees were considerably more focused on performance and seemed more disposed towards small "tricks" that could be used to improve performance at the cost of readability or conciseness, whereas the other two interviewees—both of which had PhDs in computer science—were more focused on the proper and formal ways to accomplish tasks. The two PhD-educated interviewees were

also considerably more hesitant to make general statements, so more of an effort should have been made by the interviewer to stress that all information is only expected to represent their own experiences and opinions. Points which did repeat between interviews were also clear. For example, the lack of test coverage tooling was mentioned by every developer, and two developers asserted that the best structure for transformations to take is when their abstract mappings reflect the abstract classes present in the metamodel.

Nearly word-for-word transcriptions were created for each interview, requiring[1] approximately 2 times the time of the original audio recordings and resulted in approximately 8 pages of dialog per interview. To clarify all of the points where there was confusion, contradiction or other additional elaboration would be helpful while writing the transcriptions, a set of follow-up questions was sent to each interviewer by email. On average there were four follow-up questions sent to each interviewee.

### 5.1.1 Post-processing

The transcriptions then underwent post-processing. A document of key points was created for each interview. Each key point was placed in one of five categories: context and current practices, positive reflections, difficulties, wishes, and best practices. These categories help not only organize the interview information, but also facilitate comparison between the interviews. The *constant comparison method* (used again while constructing the quality model in Chapter 6) was then used to combine interview information: Each of the key point documents were aggregated into a single document by iteratively grouping similar points together, while always maintaining backtraces to the original interviewee. The result was a single document containing a summary of all qualitative information collected during the interviews. We do not present this document as-is, but instead present the best practices and difficulties in Sections 5.1.2 and 5.1.3, respectively, which were the most useful contribution of the qualitative interview results to the quality model.

### 5.1.2 Best Practices

As noted earlier, one of the major benefits of performing interviews is that insights can be gained that uncover years of experience. Due to its rich nature, some of this material can best be presented as guidelines rather than measurable attributes. Although we have not validated this list of guidelines, we feel that many could be useful for future work or for suggestions for practitioners to build on in developing their own guidelines for QVTo. Some of these could also be formalized as design patterns. Note, however, that some may only apply to the Eclipse QVTo implementation. The best practices, which are paraphrases of developer statements resulting from the constant comparison method, are presented below in categories. More information on the QVTo language is presented in Section 2.2.

#### Variables and parameters

- Avoid mappings with arguments to increase both performance and readability. Instead, return an object and then you perform an assignment.

[1] Likely thanks to fast typing speeds

- Use global variables for data used by multiple rules (to improve understandability) as well as to replace large parameters (to improve performance).
- Use intermediate properties when you need extra information in your source model, but consider first whether it makes more sense to break up the transformations into two transformations.

#### When to use helpers, queries or mappings

- Use more mappings when possible since they do more for you transparently, such as object creation and storing trace information. The more mappings the more it looks like QVTo-style.
- Navigation over models should be separated from the mappings and placed into queries.
- Using helpers when you don't need trace information can yield better performance than mappings. Mappings however are cached, so if computing with the same parameters, they can be more efficient.
- Change helpers to queries when possible (i.e. when side-effects can be avoided).

#### Imperative vs declarative style

- Use as few `forEach` loops as possible. Generally these loops can be replaced by a new mapping or OCL (e.g. select, exist statements), greatly improving understandability and conciseness.
- Most transformations can be entirely declarative, others usually about 80% declarative and 20% imperative.

#### Transformation design

- When you need to assign an element to multiple sets, use a constructor rather than a mapping.
- Use `init` sections only when necessary. For example, if you have an abstract type and need to select a concrete type, or you need to check another object's existence. Don't use `init` sections to fill object properties.
- Inverse lookups are useful when you have complex mappings which would otherwise require multiple sweeps to gather the needed information.
- Nested `if` statements and long chains of function calls hurt clarity.
- Use inheritance to decompose the transformation into smaller problems. The mapping inheritance hierarchy should follow either the source or target metamodel.
- Use libraries to encapsulate lots of queries specific to a metamodel.
- If using mapping inheritance where the base class from a metamodel is not supported, add an `assert` in the parent mapping.

- Typically transformations begin with the `main` method, followed my the first mapping, with general queries at the bottom. Keep related mappings very near each other.
- To improve performance, one can pre-process parameters by converting them to dictionaires (though this hurts understandability).
- Using the trashbin pattern, in which objects are assigned to a dummy parent object before deleting, greatly increases performance.
- Overloading can make it unclear which mappings are executed when, so use it with caution.

#### Process

- Traces can be useful if you perform incremental transformations, particularly with large input models and don't want to regenerate the entire target model.
- If you are making in-place updates only (as opposed to model-to-model), consider graph transformation languages instead of QVTo.
- In developing transformations, start with something small and add functionality incrementally, always having a working base.

#### Syntax/Formatting

- Keep function short, with a general guideline for mapping methods to not longer than 20-30 lines of code, and especially not longer than one's screen.
- Code should be properly commented.
- Appropriate and explicit function names are important.
- At least a high-level description of what the transformation does should be at the top.
- Use consistent strictness level for OCL statements (e.g. using `select` versus `xselect`).

#### Testing

- Unit tests should focus on a specific part of the translation or a corner case, mimicking the code paths.
- Aim to have 100% code coverage, and in general more unit testing is always better.

### 5.1.3 Difficulties

We also present the difficulties mentioned during the interviews. These may be useful in informing directions for future work, both for researchers and practitioners. As with best practices, this list is not validated.

### Debugging

- Input models are often large and hard to inspect manually, especially when they contain so much backtracking information.
- Large input models in unit tests make creating expected output models very difficult.
- The source-level debugging cannot step into imported transformations or chained transformations, so log statements are in fact used more frequently than the built-in debugger.
- The debugger also has problems with multiple input models.

### Testing

- The new version of EMFCompare (2.x vs. 1.3) does not work very well when there are differences between models, yielding many false positives, possibly due to the lack of model element IDs in the CARM models.
- Many bugs are results of incomplete specifications from the start. Furthermore, the pre- and post-conditions on transformations should be made more explicit during development.
- Ensuring code coverage is currently not feasible and the testing framework used by the CARM team should be refactored.

### Code construction

- Mappings with arguments and deletion are both surprisingly slow.
- Built-in tracing from mappings can be inefficient since it is not always needed.
- Metamodel/transformation coevolution is difficult.
- There is a lot of duplicate code in the transformations being developed.

### Syntax

- There should be warnings when accidentally using = instead of := .
- There should be warnings when accidentally reassigning elements (in which case a constructor should be used, as given in Section 5.1.2).
- It's not clear what the use case for `end` blocks is.

**Environment**

- Information is hard to find and documentation is poor.
- There is not much tooling available for QVTo and existing tools often have bugs.
- Better editors are being developed, but since they are based on OCL 4 rather than OCL 3, they will not work with QVTo yet.
- The specification that queries do not have side-effects is not enforced. In general there are many points where QVTo or other tooling deviates from the specification.
- Java black-boxes do not work when calling a transformation standalone.
- You can add extra parameters from Java that don't work from the debugger window, often resulting in confusion.
- Sometimes there are bugs in the "short" notations supported in QVTo while the long notations work.
- The current limit for recursion (10,000 calls) is too low.

**Language**

- The lack of encapsulation (defining public/private) leads to unnecessary dependencies to other mappings and functions
- QVTo lacks functional programming capabilities, such as passing a function as an argument, making code more verbose.
- The semantics of mapping inheritance are not always clear. For instance, it is confusing that you can not reuse `init` clauses from other mappings even with available reuse mechanisms.
- Performance is a huge issue.

## 5.2 Review of Existing Material

Here we present a selection of our review of existing material, collected according to the approach described in Section 4.3. Although we do not present all of the results of the review, this section gives a comprehensive, if overwhelming, picture of the rich information gathered during our review. This review could serve as an additional contribution of this research. Reflecting the search terms used during this review, we organize this information in the following subsections: quality recommendations, metrics, techniques, tooling, and difficulties. Literature and the informal sources are reported together.

Note that during this review, information even indirectly related to quality was collected, since the current research specifically on model transformation quality is still quite limited, especially for QVTo itself. All information presented here was taken into account when building the quality model, discussed in Chapter 6.

### 5.2.1 Quality recommendations

Since the material reviewed related to quality was extensive, we further subdivide this section according to technology.

#### 5.2.1.1 Model transformation languages

Voelter [56] studied best practices for designing DSLs. Although our focus is not on language design, some insights are still applicable. For instance, the author emphasizes the how essential it is to have good tooling for MDE, including metamodel-aware IDEs for editing both models and transformations. Rather than proving a system is semantically equivalent to a generated system, the author finds that it is often sufficient to perform more testing rather than formal proofs, and that these tests should not test the generated code, but rather test the code execution. He also stresses the importance of good documentation. He then notes some current problems with DSLs, for example mixing notations (e.g. graphical and textual representations) is difficult to understand, tooling for migrating models and transformations after metamodels change would be helpful, and that debugging on a concrete model level is very difficult.

Lehrig [26] investigated project scenarios to aid in choosing a transformation language (QVTo, QVTr, or Java) which would lead to the highest quality end result. They did this by developing a tool to measure transformation metrics and correlated them with expert code ratings for six quality goals. They found that matching transformation functions to metamodel constructs one-to-one, few function parameters, breaking large rules into smaller ones, and use of hierarchy in rules all increased modularity. Global variables, interdependent rules, many calls to subrules, and blackboxes, on the other hand, decreased modularity. Abstracting parts of transformations and using inheritance increased reusability of solutions, whereas hard-coded model access decreased it. The availability of trace models and especially debuggers as well as documentation increased analyzability. Decreasing analyzability were large files, bad documentation, using `when` clauses since it requires reading the function signature backwards, missing pre/post conditions, too much infrastructure code, and complex flows or hierarchy in rules all decreased analyzability. To increase modifiability, few files/functions and addressing each metamodel element in a separate rule were recommended. Decreasing it were using external libraries, allowing side-effects in functions, and lack of unit testing. For consistency, following naming conventions, including code comments, and being consistent among transformation implementations increased it. To increase learnability, increased usage of OCL, minimizing infrastructure code, and improving the debugging environment were noted. Lack of documentation and inconsistent ways of implementing transformations decreased learnability.

Like Syriani discussed in Section 3.2.3, Ergin [47] argues for increased attention to design patterns in model transformations. Ergin, like this research, utilizes the MDE quality framework, in an attempt to identify design patterns and placing those patterns in a design pattern formalism, arguing that since the patterns improve quality metrics, using them more heavily in general should increase overall quality. In addition to the flattening and refinement patterns identified there, Agrawal [57] identified three other patterns for model transformation, such as computing the transitive closure in a hierarchy, and Iacob, Steen, and Heerink [58] identified five more patterns. Gniesser [59] discusses identifying and automating refactoring in ATL transformations. A number of patterns

were identified as potential targets for refactoring, partially based on Fowler's bad code smells [60]. Additional refactorings were identified such as merging helpers and extracting parent rules[2].

Rose et al. [61] looked at the results of the 2010 Transformation Tool Contest where tools were submitted which aid in model migration. Most relevant to transformation quality, they note that in particular visual debuggers and metamodel difference visualizers were considered very valuable assets by model transformation experts. We also performed our own review of the 2013 Transformation Tool Contest [62], where contestants submitted solutions to transformation scenarios. There, we extracted the following recommendations from the raw judge feedback: good white-box test cases are important, speed is important for both code and test cases, modularity and abstraction are important but cohesion within files must also be maintained, static type-checking of transformations is useful, combining graphical and text programming is hard to understand, complex dependencies should be avoided and rules should have a clear structure and application sequence, large blocks of code should be avoided, high ratios of helper code to actual transformation code should be avoided, naming and proper comments are important, debugging and refactoring support in the IDE are essential, visualizations of transformation structure are useful, and the ability to check confluence and termination are nice features of a language or IDE.

Kusel [63] performed a study on reuse amongst the sample transformations in the ATL zoo [64] collection, finding that many of the reuse concepts from object-oriented programming (OOP) are also utilized in ATL. This suggests that code duplication as well as long code blocks should be avoided by developers in much the same way as for OOP. Van Amstel et al. [65] compared performance between ATL, QVTr, and QVTo transformations. There they found that at least in ATL, declarative style programming is faster than using the equivalent imperative constructs. This could suggest that using `forEach` loops when not needed may hurt performance in QVTo.

Van Gorp [21] and Mens also make a number of suggestions about practices which benefit transformation quality. First, the authors state that a language's ability to support transformation reuse, for instance through grouping or composition, can improve a transformations readability, modularity, and maintainability. The authors also mention the importance of verifying correctness of a transformation, both syntactically and semantically, checking termination, and verifying confluence, a property of declarative languages where the output is independent of statement execution order. Therefore termination and confluence, as well as semantic verification (i.e. unit testing) may all be important for transformation quality. The authors also assert that one of the reasons declarative approaches are attractive is that they more easily accommodate model traversal and traceability management. Therefore increased declarative style in QVTo may also desirable.

A small amount of information was also found for quality recommendations for QVTo specifically. In Guduric, Puder, and Todtenhöfer's comparison of QVTo and QVTr with respect to implementation techniques [10], the author mentions a number of reuse facilities unique to QVTo which can improve quality. Specifically, large files should be split up using the reuse features of QVTo, in particular by placing helpers which are reused in libraries. We see this suggestion reflected also in a forum thread requesting expert advice about QVTo project organization [66]. There, above all else, making heavy use

[2] ATL rules are similar to QVTo mappings.

of libraries was recommended. In the same thread, however, users also caution that for performance reasons, it may be best to have as few files as possible, since the current QVTo builder is not very scalable in that regard. Therefore reuse mechanisms should be used to increase code quality, but there is currently a tradeoff with performance. In a blog post discussing the best language features of QVTo [67], the author singles out black-boxing, libraries, intermediate data, mapping extension, and transformation chaining. Since a developer is advocating these features, it suggests that using them (properly!) could lead to higher-quality transformations.

### 5.2.2 Metrics

A number of related work considers quality metrics. Van Amstel identified metrics for ATL and ASF+SDF [42]. For ATL, 36 metrics were defined and then correlated with expert ratings of 6 quality goals. Metrics which had significant correlations with at least 3 quality goals were: number called rules, number unused called rules, number parameters per called rule, unused parameters per called rule, number unused helpers, rule fan-in, unit fan-in, unit fan-out, number of units, number input models, and number output models. Metrics which were interestingly not significant were number variables per rule, number rules with `do`-sections (similar to QVTo's `forEach`), number of helpers, and helper cyclomatic complexity.

Kapova et al. [68] presented maintainability metrics for QVTr along with a discussion of the expected impact on maintainability. The metrics were identified based on previous literature on GPLs as well as the authors' experiences. The metrics included transformation size metrics (e.g. lines of code, number `where` clauses), relation metrics (e.g. dependency fan-in and fan-out), consistency metrics (e.g. number code clones), inheritance metrics (number parents [where more inheritance is expected to be *less* maintainable), an manual metrics (e.g. number of relations following a design pattern, similarity of relations).

Kolahdouz-Rahimi et al. [27] evaluated model transformation approaches for refactoring, following the GQM framework. Although their purpose is comparing different approaches (i.e. transformation languages), still some insights are applicable to transformation quality. Like in [42], experts rated code samples which were correlated with metrics. They found that OCL notation helps with comprehendability of ATL transformations, but also that conciseness and complexity are often a tradeoff in transformation implementations. They also found that complexity and size correlate positively with development effort and negatively with modularity and usability, suggesting that low complexity and small size may lead to higher-quality transformations.

For QVTo specifically, Nguyen [69] identified 60 quality metrics and developed a tool to measure them on simple QVTo transformations. These metrics included number assert calls, cyclomatic complexity, number resolve expressions, number variables per mapping, and number imported modules. However, no evaluation is performed and the tool is not integrated into an IDE or workflow.

### 5.2.3 Techniques

Techniques to improve quality are also considered in related work. Planas, Cabot, and Gómez [70] present two correctness properties for ATL: executability and coverage. Satisfying executability states that every rule can be executed, taking into account the potential constraints (e.g. `where` clauses) placed on models and rules. To satisfy coverage, the entire source and target metamodels should be addressed by the transformation. Therefore the authors assert these properties should be satisfied before a transformation is considered correct. Both properties also suggest that dead code (in either transformation code or model code) can hurt correctness.

Selim, Cordy, and Dingel [71] describe the state of the art of testing model transformations, focusing primarily on coverage adequacy criteria (for both white-box and black-box testing) and test case generation. McQuillan and Power [72] elaborate on white-box coverage criteria for model transformations, which combines metamodel coverage, grammar coverage, and transformation code coverage. They then applied their coverage approach to ATL transformations from the transformation zoo.

Van Amstel and van den Brand [73] presented three analysis techniques that can be used to improve maintainability of ATL, QVTo, and Xtend transformations. These techniques were metrics (based on [42]), structure and trace analysis, and metamodel coverage analysis. Van Amstel, van den Brand, and Serebrenik also investigated using traceability visualizations for model transformations, describing their technique using higher-order transformations. They found that such visualizations helped with debugging as well as impact analysis.

### 5.2.4 Tooling

Stahl and Völter [15] notes first that the OMG is not very strict about standards compliance of implementations, which leads to poor consistency between tools. Rentschler et al. [74] have developed a tool for interactively visualizing dependency analysis in transformations as an Eclipse plugin, performing a small empirical investigation demonstrating that it helped developers navigate through transformations. The tool is available freely online and works with select versions of Eclipse. In [11] Ciancone, Filieri, and Mirandola present an approach for QVTo model transformation testing in which the test cases themselves are designed within QVTo, arguing that this eliminates the need to move between development environments and is faster than XPath- or OCL-based approaches. Although the download links for the tool were not functioning, a copy of the tool was obtained from the authors directly.

Paige and Varró [75] described their lessons learned while creating MDE tooling for ten years. Their experience with respect to quality assessment and improvement was that rich graphical views of models are more useful than editing tools, that it's hard to balance usability and performance, design tasks should not be overautomated, it's important to see what user priorities are before spending effort on secondary features, and that developer productivity tools and documentation are very important.

In addition to reviewing ASML materials, such as lecture materials about the TIOBE TICS tool for C and C++ code quality, we also spoke with the ASML quality assurance employee who introduced the TIOBE TICS tool at ASML. According to him, being able

to measure and visually browse these metrics within the code base helps component owners improve quality in their own code. He stressed however that the primary users of the tool should be component owners themselves, rather than another entity reviewing metrics and setting metric goals.

### 5.2.5 Difficulties

Finally, we review the existing material related to current difficulties in developing QVTo transformations. The best source of information here was the online forum [76]. There, users had recurring trouble with URI resolution, performing cross-project builds (i.e. using files from another project), using multiple input/output models (e.g. figuring out the correct syntax), (lack of) support for UML stereotypes, getting Java blackboxes working properly, syntax differing from the specification, and general lack of documentation.

## 5.3 Introspection

The final component of the exploratory study was introspection, namely learning QVTo by the author using the Eclipse QVTo implementation. We discuss the most relevant notes made during this process. First, there was a strong tendency to create large `init` sections inside mappings, manually constructing the output object, since this was more similar to GPL programming. However, in every case it was possible to refactor to use only the `population` section once more comfortable with the syntax. There was a similar tendency at first to create many local variables, when instead a more complex (but still quite readable) OCL expression could be used, resulting in a much more concise mapping. Configuring the URIs for the metamodels also proved quite challenging, where even the most comprehensive tutorials (e.g. [55]) are not up-to-date with the optimal ways of doing this.[3] It was also very unclear what the use cases for helpers was, since all examples followed here could be implemented without side-effects in queries. As far as we could tell, it also seems that mappings cannot accept sequences as input (although sequences too should be considered an object type). Instead, queries or helpers had to be used. This leads in turn to overall more lines of code and loss of traceability that could be useful for debugging.

There were also some issues with the syntax, for instance we could not figure out the correct syntax for inline object-instantiation, leading again to more verbose code. The syntactical difference between assignment (:=) and boolean equals (=) is also confusing, since in the current implementation there are no warnings if one uses the latter accidentally. Finally, assignment to a multi-valued object in QVTo seems to work using either single values or multiple values on the left-hand side (e.g. `my_list := an_object` and `my_list := some_objects` both compile, in which case the former is converted to a sequence), which could be confusing if done accidentally. Therefore avoiding this syntax for single objects may be preferable.

[3] In the case of [55], we contacted the tutorial author once we had discovered the correct way to configure the URIs without errors appearing in the IDE, some two months later.

# Chapter 6

# QVTo Quality Model

In this chapter we present the quality model created from the material gathered during the exploratory study. First we describe how we constructed the model in Section 6.1. The model itself is presented in Section 6.2 and discussion is in Section 6.3.

## 6.1 Constructing the Quality Model

Presented in Chapter 5 was a huge amount of qualitative data, ranging from scientific literature on software quality to interviews of QVTo experts. To be able to use that data to assess QVTo quality, we formalize it into a coherent quality model, in the form of the sample quality model presented in Section 3.3.2. This formalization is namely steps 1 through 5 of the MDE framework.

Step 1 of the MDE framework is identifying quality goals. Do this, we again incorporated what is known is classical theory generation as the constant comparison method [52], as used already for aggregating the interview best practices and difficulties in Section 5.1. Specifically, all qualitative data from each of the three exploratory study components was translated to sets of *key points* related to transformation quality, approximately one sentence per point. In a process similar to what is sometimes called *coding* [52], we then *tagged* each point with keywords describing why this key point was important for quality (e.g. "conciseness"), matching the wording used in the raw data as closely as possible. Then an iterative approach was applied where similar points are grouped together, but always maintaining traceability link back to the original data source. When two points contradicted one another, they were still combined but the traceability link was marked as negative support. This resulted in a large list of key points with many unique tags. In particular, readability and performance were mentioned frequently. The set of quality tags was then reduced by combining closely related tags, replacing them with a more general term that described both. These tags comprise the most relevant quality goals for QVTo.

Step 2 of the MDE framework is specifying target objects. We assigned each of the key points formulated earlier to the target object it represented. Target objects considered were: transformation model, transformation process, metamodels, QVTo language, development process, and developer tooling. After assigning target objects, it was clear

that transformation model was overwhelmingly the most important. This is not surprising, since our focus on internal quality is most related to the transformation itself. An example of another target assigned to a key point was the development process. The key point assigned there was "having too many tools required to develop a transformation indicates a less mature development process", since interviewees noted that it was cumbersome using multiple Eclipse instances for development and since literature suggested that poor tool integration leads to worse developer experience and productivity. Assigned to the QVTo language target was for example one interviewee wishing there were more functional language constructs. At this point, it was then chosen that because the other targets were mentioned considerably less prominent as well as less relevant to internal quality, we would focus our efforts exclusively on the transformation model target in our final quality model.

Step 3 of the framework is to identify the quality attributes for the transformation model target. For this we used the key points. Since a quality attribute is better a short phrase than a sentence (per the principles of quality model frameworks in Section 3.3), we translated each point to a short phrase. To give an indication of the nature of this phrase, specifically whether it helps or hurts quality, we also gave a *directionality* to the phrase. The quality goals associated with the point were also marked with a direction. For example, "Using mappings instead of helpers increases understandability, but can hurt performance due to the additional tracing added by the engine" was converted to "More mappings than helpers" with goals "Understandability (+)" and "Performance (-)". At all times the traceability links were still maintained to the original sources. The resulting list of phrases is then the quality attributes of our quality model. Notably, very vague qualifiers such has "effective" or "appropriate" were avoided in the attributes, favoring instead specific directions (e.g. "more", "few") since they are easier to interpret. The directionality was assigned so that *in general*, the more a transformation achieves the attribute, the higher quality it is.

For step 4, evaluation procedures were added for each quality attribute. These evaluation procedures were based on a combination of methods suggested from literature or during the interviews and methods thought by the first author to sufficiently measure the attribute, according to her experience in QVTo. Finally, step 5 requires links to be made between the quality goals and quality attributes. These links are already present thanks to our constant comparison method and tagging process. No additional links were added.

After this step the quality model is complete, presented in Section 6.2. The remaining two steps of the framework are evaluation and implementation. Evaluation is covered in Chapter 7. Implementation is the subject of Part II.

## 6.2 Resulting Model

Our QVTo quality model is presented in Table 6.1, containing quality attributes, quality goals, and evaluation procedures. Because all attributes are for the transformation model target, the target column has been excluded from the table. The traceability links are also not present because the majority was discussed in detail in Chapter 5. The quality model consists of 37 quality attributes and 4 quality goals, namely Functionality, Understandability, Performance, and Maintainability. Although the list of attributes may seem large, our evaluation (Chapter 7) distinguishes their relative importance.

| Quality attribute[1] | Quality goals | Evaluation procedure |
|---|---|---|
| | QVTo | |
| Deletion uses trashbin pattern | P+ | # Instances where trashbin pattern not used |
| Few blackboxes | U+, M+ | # Blackboxes ] |
| Few configuration properties | U+, P- | # Configuration properties |
| Few intermediate properties | U+ | # Intermediate properties |
| Few **end** sections | U+ | # **end** sections |
| Few queries with side-effects | U+ | # Queries with side-effects |
| Few **when** and **where** clauses | U+ | # **when** and **where** clauses |
| Little imperative programming | U+, M+ | # **forEach** loops |
| Mappings only used when tracing needed | U-, P+ | # Instances when mapping used but not tracing |
| Minimal reassignment of objects | F+ | # Instances when object assigned to multiple sets |
| More mappings than helpers | U+, P- | Ratio # mappings to # helpers |
| More queries than helpers | U+, P+ | Ratio # helpers to # queries |
| Small **init** sections | U+ | LOC inside **init** sections |
| | MMT | |
| Confluence satisfied | F+ | Proof of confluence |
| Few input/output models | U+, M+ | # Input/output models and metamodels |
| Inheritance usage matches metamodel | U+ | # Abstract classes in common with metamodel |
| Pre- and post-conditions specified | F+, M+ | Presence of formal specification |
| | GPL | |
| Detailed comments throughout code | U+ | Comment/LOC ratio |
| Few dependencies between functions | U+, M+ | Function fan-out within module |
| Few dependencies on other modules | U+, M+ | Module fan-out |
| Few mapping arguments | P+, U+ | # Arguments per mapping |
| Few nested **if** statements | U+ | Nesting depth |
| Formatting conventions followed | U+ | # Violations of a coding standard |
| High test coverage | F+, M+ | Test code coverage |
| High usage of design patterns | U+ | Comparison of patterns used to a pattern catalog |
| Interdependent functions near each other | U+, M+ | Custom semantic similarity measure |
| Little dead code | U+, M+ | # Unused LOC |
| Little overloading | U+ | # Instances of overloaded functions |
| Low code duplication with other modules | M+ | # Instances where at least five lines repeated |
| Low code duplication within module | U+, M+ | # Instances where at least two lines repeated |
| Low execution time | P+ | Execution speed with typical model |
| Low syntactic complexity | U+ | Syntactic complexity measure [27] |
| Short function chains | U+ | Length of chains |
| Small function size | U+ | Function LOC |
| Small transformation size | U+ | Module LOC |
| Small interfaces to other modules | U+, M+ | Function fan-out to other modules |
| Termination checked | F+ | Proof of termination |

TABLE 6.1: **QVTo quality model resulting from the exploratory study.** Contains quality attributes, quality goals (F, U, P, M for Functionality, Understandability, Performance, and Maintainability, respectively), and evaluation procedures. Attributes are organized according to whether they are best described as specific to QVTo, MMT, or applicable also to GPLs and then ordered alphabetically.

The attributes have been classified according to whether they can be best-described as specific to QVTo, MMT, or neither, in which case they would also be applicable also to GPLs. Although these classifications are up to interpretation, it provides an overview of the proportion of attributes which are specific to each context. For example, "Mappings only used when tracing needed" is unquestionably QVTo-specific. However, we have also placed "Deletion uses trashbin pattern" under QVTo-specific, because the reason it is included in the model is related to the implementation of the QVTo engine. According to our classification scheme, 13 attributes are QVTo-specific, 4 are MMT-specific, and 20 are applicable to GPLs.

## 6.3 Discussion

It is clear from the quality model that understandability and maintainability were the most ubiquitous (though not necessarily most important) quality goals for QVTo practitioners. The first reason for this is our focus on internal quality, which is more dependent on maintainability and understandability than on functionality or performance. A second reason is likely that understandability and maintainability are more complex to describe, therefore requiring more attributes. Furthermore, developers themselves have a high amount of control over the attributes affecting these goals, whereas for others (for example performance, which can be heavily engine-dependent) may allow less control. Therefore a model containing many attributes corresponding to understandability and maintainability could prove more valuable in helping developers assess their QVTo transformation quality.

It is notable then that despite our focus on internal quality, the quality goal Performance as well as the attribute "Low execution time" are still included, since these may be more closely related to external quality (quality in use) rather than internal quality. However, it is present in our quality model because it was clear from the exploratory study and in particular the interviews that a description of QVTo quality without a consideration for performance would feel incomplete, due to the importance of good performance for QVTo transformations in practice (and in particular for the CARM team). Therefore, according to our pragmatist stance, it is still included in our quality model.

The quality attributes can also be categorized according to their nature, as shown in Table 6.2. Two of the attributes can be considered presentation-related, since they focus on style and are unrelated to the behavior of the transformation. Nine of the attributes are related to high-level architecture, for instance how to organize your transformations in a project. Four attributes are related closely to the current engine implementation, and therefore could change with future implementations or implementations other than Eclipse. The remaining 22 attributes can be considered quality attributes local to a transformation, i.e. very specific to a how a single transformation has been written. Categorizing the attributes according to their nature in this way helps determine when and where each attribute could be applied. For example, if a developer has access to improve a transformation but not able to tackle refactoring the architecture of her project, then the transformation-local attributes may be most relevant. As another example, if a new version of the QVTo engine is released, it may be necessary to reevaluate the implementation-dependent attributes.

That the model was built bottom-up can also be seen. For example, the attribute "Small `init` sections" is included, while for `end` sections, "Few `end` sections" is included. This reflects that according to our exploratory study, the *amount* of code inside `init` sections affects quality, whereas for `end` sections it was only suggested that the *frequency* of use may affect quality. Recall also that no direct attempt was made to cover QVTo language features in the exploratory study, so the QVTo-specific attributes included the model are there because they emerged as relevant to quality during the triangulation approach.

| Presentation | Transformation local |
|---|---|
| Detailed comments throughout code | Confluence satisfied |
| Formatting conventions followed | Few dependencies between functions |
| **High-level architecture** | Few `end` sections |
| Small transformation size | Few mapping arguments |
| Pre- and post-conditions specified | Few nested `if` statements |
| Few dependencies on other modules | Few `when` and `where` clauses |
| Small interfaces to other modules | High test coverage |
| Low code duplication with other modules | High usage of design patterns |
| Few intermediate properties | Inheritance usage matches metamodel |
| Few blackboxes | Interdependent functions near each other |
| Few configuration properties | Little dead code |
| Few input/output models | Little imperative programming |
| **Implementation-dependent** | Little overloading |
| Few queries with side-effects | Low code duplication within module |
| Low execution time | Low syntactic complexity |
| Deletion uses trashbin pattern | Minimal reassignment of objects |
| Mappings only used when tracing needed | More mappings than helpers |
| | More queries than helpers |
| | Short function chains |
| | Small function size |
| | Small `init` sections |
| | Termination checked |

TABLE 6.2: Nature of attributes included in the QVTo quality model

### 6.3.1 Conformance to ISO/IEC 25010

As an international standard, ISO/IEC 25010 represents a broad consensus for how to describe software quality. It is therefore valuable to show that our QVTo quality model conforms to the software product quality model. According to the standard, conformance can be shown either by using the quality model already provided by the standard from the start, or by demonstrating the traceability links between the standard model and the tailored model. We demonstrate conformance using the latter method, namely by specifying links between the software product quality model and our QVTo quality model.

First we map our quality goals to the quality characteristics of the standard: Functionality is mapped to Functional suitability, Performance to Performance efficiency, Understandability to Usability, and Maintainability to Maintainability. In each case the reason for using a different name in our model is that these terms were more natural for developers. Notably, our model excludes the characteristics Compatibility, Reliability, and Security, and Portability. They are excluded since, according to our exploratory study, they are lesser concerns in QVTo development at this time. Finally, our quality attributes can each be mapped to a single quality property in the standard quality model, but without explicit directionality since quality properties do not indicate a direction. For example, "Few input/output models" is mapped to "Number input/output models".

### 6.3.2 Similarities to other models

Our QVTo quality model is most closely related in purpose to the model transformation quality metrics proposed by van Amstel and Nguyen [6], mentioned in Section 3.2. There, metric sets for QVTo, ATL, and Xtend were defined. Coincidentally, the metrics set for QVTo also contains 37 items, like our set of quality attributes. There, the authors have

categorized the metrics according to notions similar to our quality goals. The categorized metrics are shown in Table 6.3, along with the similar attributes from our model. When a quality attribute is similar to more than one metric from [6], it is shown in italics.

Of our 37 attributes, we see that 13 are similar to metrics from [6], approximately one third. We also see that similarities appear in each of the metric categories identified by [6]. This is significant because their metrics were constructed in a top-down fashion by the authors, based only on theory. That there is overlap suggests that our synthesis approach also manages to span a similar range of concerns as the top-down approach.

The metrics which overlap, however, comprise the more straightforward attributes from our model, such as "More mappings than helpers" and "Few input/output models". Our more complex attributes (e.g. "Deletion uses trashbin pattern", "Interdependent functions near each other"), on the other hand, do not have counterparts in the metric set. In some cases, a metric is similar to an attribute, but as a result of the bottom-up approach, the attribute is substantially more nuanced. For example, while the model from [6] includes the metric "# Abstract mappings", ours includes the attribute "Inheritance usage matches metamodel" because according to our study it was not simply the number of abstract mappings that affected quality, but the extent to which they represent the abstract classes in a metamodel. Notably, a number of attributes identified as important during our exploratory study are not present in [6]. For example, "Little imperative programming" was recognized in each of our developer interviews in addition to some literature as important for maintainability and understandability, but has no counterpart in the metrics from [6]. Our model also includes more QVTo-specific attributes (e.g. "Few `end` sections"). These may be excluded from [6], however, because there are simply too many language-specific constructs to add an attribute for each construct. This difficulty to distinguish the most important features of new domain is in our opinion a fundamental problem with theory-based approaches.

Van Amstel's later work presenting a quality model for model transformations addressing primarily ATL and ASF+SDF [42] (discussed in Section 3.2.2) builds on [6], adding metrics such as "Depth of inheritance tree" for ATL. Still, however, due to the theory-based approach followed, the metrics still describe relatively shallow properties in comparison to the rich data incorporated in our quality attributes, thanks to our synthesis approach.

Kapova et al. [68] (mentioned during our existing material review in Section 5.2.2), presented a set of maintainability metrics for QVTr. There a combination of "automated" and manual metrics were provided. The automated metrics, like those from [6] and [42], are quite simple, including "# Local variables" and "# `when` predicates". The manual metrics, however, contained "Similarity of relations"[2], "# Relations that follow a design pattern", and "Type cut through source/target metamodel". The last in particular is similar in essence to our "Inheritance usage matches a metamodel" attribute, since it measures the match between metamodel elements and the elements addressed by relations. The metric is presented however only in the context of increasing metamodel coverage (chosen as a quality goal by the the authors), rather than contributing to understandability like our attribute. Because our attribute was voiced as important by multiple interviewees for transformation readability and understandability, we consider it an important attribute of our model. Therefore, while related quality models utilized theory-based approaches, we strongly advocate the use of synthesis approaches in future quality research.

[2] QVTr *relations* are similar to QVTo mappings

| Category | Metric | Similar quality attributes |
|---|---|---|
| Size metrics | # Mappings | *More mappings than helpers* |
| | # Helpers | *More mappings than helpers*, More queries than helpers |
| Function complexity metrics | # Parameters per mapping/helper | |
| | # Variables per mapping/helper | |
| | # Operations on collections per mapping/helper | |
| | Mapping/helper cyclomatic complexity | Few nested `if` statements, Low syntactic complexity |
| Modularity metrics | # Modules | |
| | # Imported modules | *Few dependencies on other modules* |
| | # Times a module is imported | |
| | # Mappings/helper per module | Small transformation size |
| Inheritance metrics | # Abstract mappings | Inheritance usage matches metamodel |
| | # Mapping inherits | |
| | # Mapping merges | |
| | # Mapping disjuncts | |
| | # Mappings per disjunct | |
| | # Overloaded mappings/helpers | *Little overloading* |
| | # Mappings per mapping name (overloadings) | *Little overloading* |
| | # Helpers per helper name (overloading) | *Little overloading* |
| Dependency metrics | # Calls to mapping per mapping | *Few dependencies between functions* |
| | # Calls from mapping per mapping | *Few dependencies between functions* |
| | # Calls to helpers | |
| | # Calls from helpers | |
| | # Calls to mappings/helpers in other modules | *Few dependencies on other modules* |
| | # Calls from mappings/helpers in other modules | Small interfaces to other modules |
| Consistency metrics | # Unused modules | *Little dead code* |
| | # Unused mappings | *Little dead code* |
| | # Unused helpers | *Little dead code* |
| | # Unused parameters | *Little dead code* |
| | # Unused local variables | *Little dead code* |
| | # Variables with same name but different types | |
| | # Calls to `log` | |
| | # Calls to `assert` | |
| Input/output metrics | # Input/output models | *Few input/output models* |
| | # Imported metamodels | |
| | # Input models per transformation | *Few input/output models* |
| | # Output models per transformation | *Few input/output models* |
| | # Imported metamodels per module | |
| QVTo-specific metrics | # Intermediate class/properties | Few intermediate properties |
| | # Trace resolution calls | Mappings only used when tracing needed |

TABLE 6.3: **Comparison with quality metrics proposed by [6].** Metric categories are specified by [6]. The most similar quality attributes from our quality model are displayed on the right. When similar to multiple metrics, the attribute is displayed in italics.

# Chapter 7

# Quality Model Evaluation

Although the bottom-up components of the synthesis approach guarantee initial empirical validity for each attribute, validation is still required for confirmation and to show generalizability. In particular, according to the pragmatist stance, validation is required to show that a particular theory does in fact work in practice [46]. First we review existing methods for evaluating quality models in Section 7.1. Then we describe the drawbacks of the validation techniques used in the most closely-related literature in Section 7.2. We describe our chosen evaluation approach in Section 7.3. Like our synthesis approach to building the model, we consider our evaluation approach to be one of the contributions of this research, as we argue it provides more convincing evidence that model attributes are useful for quality in practice. The results of our evaluation are presented in Section 7.4 and the analysis thereof in Section 7.5, respectively. We close by discussing remaining threats to validity in Section 7.6.

## 7.1 Validation Methods

Model validation can be defined as "the process of determining the degree to which a model and their associated data are accurate representations of the real world from the perspective of its intended uses" [77]. Validation is essential since theory does not always work in practice. Despite this, in the field of quality research, few attempts are made to empirically validate quality models, especially in comparison to the number of models and frameworks proposed [46].

In this section we techniques to evaluate quality models. Which discuss technique is chosen by a researcher depends not only on its applicability but also, again, depends heavily on the researcher's philosophical stance, since that stance determines what is acceptable for truth.

### 7.1.1 Validation versus evaluation

In general, a distinction can be made between the notions of *validation* and *evaluation*. According to Oreskes' [78], valid implies "being supported by objective truth." To fulfill that, she provides a number of requirements for a quantitative model to be validateable:

- The model must be observable and measurable.
- The model must exhibit constancy of structure in time.
- The model must exhibit constancy across variations in conditions not specified in the model.
- The model must permit collection of ample data.

Therefore, most models used in social settings are not, despite claims, validated, for instance because they are not sufficiently generalizable. Although we could argue that because we have adjusted our definition of truth by adopting a philosophical stance, and therefore a model may be considered entirely valid against that truth, we nonetheless choose to refer to the evaluation of our model to prevent a false sense of certainty for the reader.

### 7.1.2 Empirically evaluating quality models

Model evaluation is typically done empirically. Moody [46] elaborates on how different empirical methods can be used to evaluate quality models.

#### 7.1.2.1 Controlled experiments

Controlled laboratory experiments can be considered the most powerful form of model evaluation [46]. Laboratory settings offer an exceptionally high level of control, allowing researchers to draw relatively certain conclusions from observing a quality model in use. The downside, however, is that the only experiments possible to conduct in a laboratory setting tend to be extremely simple when compared to how a model may be used in practice. Another downside is that experiments can typically only be carried out on a small number of subjects, putting their generalizability into question.

They have however been used to evaluate models in a number of cases: A controlled experiment successfully distinguished data models made by experts and made by novices, allowing the researchers to conclude that the model produced by experts was more correct [79]. A laboratory experiment was also used to evaluate the predictive validity of a set of object-oriented quality metrics [49]. To assess a quality model for process models, a double-blind experiment was performed which provided insights into the quality model's effectiveness in evaluating the process models [80].

#### 7.1.2.2 Action research

Action research allows one to test and refine research ideas by applying them in practice. Unlike in experiments, there are no control groups and the researchers conducting the study are actively involved in the constant refinement of the model as it is being applied. The downside to action research is the lack of control, so when using it to evaluate a quality model, it is hard to be certain that whatever improvements in quality do occur are actually due to the model.

In practice, action research has been used for instance to evaluate a quality model for entity relationship models [46]. There, the quality model was used in twenty projects over the course of two years during which it was continuously refined, finally resulting in a quality model presumed trustworthy by the researchers. In another study to evaluate quality metrics, action research was used by having five projects use the metric sets, removing non-useful metrics after each project [81]. The result was a significantly smaller set of metrics which developers actually considered were useful in practice.

As a more close-to-home example, the use of TIOBE TICS [35] at ASML (as described in Section 2.3) is an example of action research, where the metrics measured by the tool are refined over time according to ASML's needs.

#### 7.1.2.3 Surveys

Survey research is used to collect data from a representative sample in order to make generalizations about a certain population [17]. According to Moody [46], surveys can be used to evaluate quality models by having people already using the model fill out a survey about their experiences with it. Since in most cases it is difficult to find a sample of people who are already using a given quality model, this method is generally not applicable.

However, a review of evaluation techniques used by other software quality researchers reveals that surveys have nonetheless been used to evaluate quality models. Van Amstel [42] evaluated his set of quality metrics for ATL by sending surveys out to 19 experts. Each expert received seven ATL transformations which they then rated on six quality goals related to maintainability (for the specific goals, see Section 3.2.2). The quality metrics were then measured on each transformation and correlated with the expert ratings for each quality goal. By measuring correlations, the link between quality goals and metrics can be evaluated. However, although some correlations were found between some metrics and quality goals, none was statistically significant due to the small sample size, often requiring conclusions to instead rely heavily on qualitative feedback. Lehrig [26] and Kapova et al. [68] use the same evaluation technique, though in the former Java, QVTo, and QVTr transformations are compared, and in the latter QVTr transformations are evaluated. Both cases however, like van Amstel, suffer from relatively low-power conclusions. Finally, Kolahdouz-Rahimi et al. [27] evaluated transformation languages also by performing such rating surveys. The ratings were then used to suggest which language may be appropriate given certain types of transformation tasks. Therefore we see that using surveys to evaluate metrics is a relatively popular evaluation method for quality models for model transformations. Surveys like these are indicative of a the positivist stance, since general knowledge is being inferred from the characteristics of a small experiments.

Voelter [56] uses surveys to perform evaluate of a list of best practices to follow when designing domain-specific languages (DSLs). There, a small survey was conducted of experts where each best practice was rated according to how much confidence the expert had in that practice. Unlike the evaluation surveys described above using correlations, the survey in [56] leverages the consensus-building techniques, therefore implicitly adopting a more constructivist or pragmatist stance.

## 7.2 Drawbacks of Approaches from Related Work

The approach used in the most closely-related work is that used in [42], [26] and [68]. As described in Section 7.1.2.3, these approaches conducted a survey where experts rated code samples on quality goals, which were later correlated with metrics. In addition to the conclusions being drawn from this validation method being in general weak, we also now simulate what this validation approach could yield if it were used to evaluate our quality model.

We observe that in this method, experts are required to make snap judgements based on a visual sample of the code. This method therefore first strongly biases judgements towards visual attributes, such as attributes related to readability. Therefore attributes related to other quality goals (e.g. termination checked) would possibly incorrectly receive lower correlations. Second, attributes which cannot be deduced visually at all (e.g. low code duplication with other modules, high test coverage, low execution time) or cannot be deduced *quickly* (e.g. few dependencies to other modules, mappings only used when tracing needed) would likely go unnoticed by the expert reviewer. Furthermore, in addition to the bias towards readability, there are simply so many metrics (in both the related work and this work) that it is not possible to control for each attribute individually, making it impossible to evaluate each metric; to do so would require preparation of an extremely large set of code samples which experts would need to review, which is infeasible within reasonable time constraints. In particular, this method would therefore yield lower correlations for less-common metrics, even if they are related to readability (e.g. small init sections).

Therefore, a large portion of our quality model would likely go unvalidated if we were to use this method, even if the attributes in fact are important for QVTo transformation quality. Encountering some of these problems with low correlations, [42] for example ended up instead relying heavily on the qualitative data offered by the expert's comments to gain insight on a more fine-grained level.

## 7.3 Our Evaluation Approach

There are a number of parts of the quality model that could benefit from evaluation: the relationship between the quality goals to QVTo quality in general, the relationship between quality goals and quality attributes, and the relationship between the quality attributes and the evaluation procedures. For each of those relationships, many questions could be asked. For instance, how *generalizable* is the relationship? Or, how *complete* is the relationship, e.g. do the quality attributes give a complete picture of the quality goals, or are there additional attributes that may be required to assess a goal? For our evaluation, we have chosen to focus on confirming the relationship between the quality goals and quality attributes. This differs from the evaluation from [42], where the relationship between the quality goals and the metrics in evaluated directly (though also in that work, attributes and evaluation procedures are not distinguished). We focus on the relationship between goals and attributes because validating the relationship between metrics and attributes without knowing whether the attributes are useful would be meaningless, and because of our synthesis approach we are more confident in the relationship between QVTo quality and the quality goals. We also focus more on correctness of the relationships than completeness, i.e. we investigate whether the goal-attribute links we

have established are correct, rather than focusing on whether there may be additional attributes to fully describe the goal.

With our assumption that developers themselves are one of the best resources to determine what high-quality code is and per our discussion of evaluation techniques from Section 7.1, we evaluate our quality model by performing a *survey of developer perceptions* of each quality attribute. Using perceptions is based on the pragmatist stance, directly leveraging practitioner experience for measuring success. Furthermore, by addressing each attribute individually, we are able to assess the model on a very fine-grained level. Therefore we address the drawbacks of the approaches discussed in Section 7.2. To assess generalizability of the quality model, both CARM and QVTo developers from the general public participated in the survey.

### 7.3.1 Survey design

Like with interviews, surveys must be scrupulously designed to make them effective and valid. Surveys can ask either closed questions (e.g. multiple choice) or open-ended questions. Since our approach to building the model utilized qualitative data so heavily, we have opted for our evaluation to support more quantitative analysis, therefore closed questions. Specifically, we have chosen to use an *attitude Likert scale* questions, where respondents rate their level of agreement with a statement [82]. An alternative to rating is *ranking*, where respondents would for instance rank the attributes by importance. Ranking surveys have been shown to at times have higher test-retest reliability, meaning if the subjects retake the survey later, the results are more similar to the first time [83]. Ranking, however, can take up to three times longer for a respondent to answer than a rating question with the same number of items and is therefore more of a burden on respondents [84], particularly unrealistic for 37 attributes. Moreover, other research has found that rating and ranking scales can be equally effective [84]. To achieve our desired granularity, we include a question concerning each quality attribute in every survey.

When opting to use an attitude Likert scale, there are a number of design considerations. First, avoid "double-barreled" questions [85]. These are questions that include two topics when the respondent may only agree with one (e.g. asking to agree or disagree with the statement "Schools that fail to attract enough pupils should be closed and the teachers lose their jobs"). Second, quantitative statements such as using the word "always" or "never" should be avoided, since they can also introduce ambiguity in the response. Finally, the survey should be *balanced* [85]. A balanced survey is one in which all questions are worded similarly. Although acquiescence bias, where respondents passively or even subconsciously indicate responses which agree with the question, is a major drawback of Likert scales, it is more important to have this bias consistently throughout the survey [86].

In addition to the question wording, the response options must also be designed. First, it has been shown that six or seven is an ideal number of response items for attitude Likert scales, where fewer options results in less rich data, and more hurts response reliability [87]. The scale then typically includes options for "Strongly disagree", "Disagree", "Somewhat disagree", "Somewhat Agree", "Agree", and "Strongly agree". Whether to include a neutral point is another consideration, since people tend to select the neutral response when they are afraid of offending the reviewer, making neutral responses harder to interpret [88].

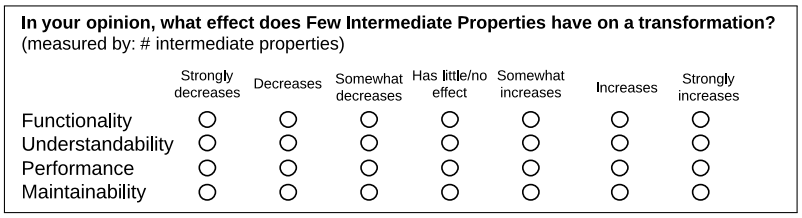


FIGURE 7.1: Example question from validation survey

Four our questions, we have opted for a *compound* Likert-style question for each of the 37 quality attributes. This question asks the respondent about the relationship between a given attribute and the four quality goals. An example attribute question is shown in Figure 7.1. The question wording was constructed so that information could be gained both on the *nature* of the relationship between the goal and attribute (i.e. positive or negative influence on a quality goal) as well as the *magnitude* of the relationship, while still avoiding double-barreled questions. A response option for "Don't know" was excluded for fear that many respondents may answer with that response simply because they feel their opinion may not be generalizable. Instead, it was assumed that if a respondent really felt uncomfortable answering a question (for example because they didn't understand the attribute) that they would add a comment, which could be processed by the researchers later. Furthermore, each question was wording using the same directionality as in the quality model itself. Therefore, we fulfill the balanced survey recommendation, and the expectation is that for each quality goal identified as relevant for the attribute, a positive (i.e. "increases") response is expected. This expectation for positive responses is also clear to survey takers, however, making it susceptible to acquiescence bias. However, this style was maintained since the consistent form of the questions makes the survey easier to complete, therefore requiring less time from the respondent and reducing chance of confusion.

The survey began with an introduction to the research. We used two different survey variants, one with a short introduction for the CARM developers who were already familiar with our research, and one with a more comprehensive introduction for non-CARM developers. Then followed instructions and a field for the developer to describe their experience with QVTo. The survey instructions stressed that the developer should answer the questions in terms of typical QVTo transformations and should reflect their own experiences. The introduction and instructions used for non-CARM developers is presented in Figure 7.2. After each question was a field for comments, and for quality attributes with less well-known terms (e.g. "confluence"), a short explanation was provided.

A mock survey was carried out with an independent developer beforehand to test the survey clarity and completion time, after which the introduction and instructions were updated. From the mock survey, the completion time for our survey was set at 25 minutes. The public version was distributed online with Google Forms by posting on the Eclipse QVTo forum [76] and the Eclipse QVTo developer mailing list [89], in an attempt to reach both QVTo users as well as the language developers. The link to the CARM survey was sent directly to the CARM developers. Survey responses were accepted over the period of one month.

**QVTo Quality Survey**

Welcome to the QVTo Quality Survey.

In this study, our goal is to understand how to assess quality of QVT Operational Mappings (QVTo) code. We do this by building consensus among experts, namely you. The next phase of the research is to develop a tool which helps QVTo developers write higher quality code by measuring or visualizing some of the attributes identified here. The quality model based on this study and any tools developed will be open sourced! Therefore your participation helps improve QVTo development and adoption in the future.

Your participation is voluntary and confidential. If you agree to participate, you will be asked to provide information on how certain attributes of QVTo code relate to quality. The audience of the study is anyone experienced in QVTo and we ask that you complete the survey only once. Participation in this study is expected to take approximately 25 minutes of your time. There are 37 questions total.

This project is being carried out by computer science researchers from Eindhoven University of Technology, The Netherlands (Christine Gerpheide <c.m.gerpheide@student.tue.nl> and Alexander Serebrenik <a.serebrenik@tue.nl>) and in cooperation with ASML N.V. (Ramon Schiffelers <r.r.h.schiffelers@tue.nl, ramon.schiffelers@asml.com>, The Netherlands, which uses QVTo in industry.

We thank you in advance for your participation in this study. We will post the quality model resulting from this survey back here after the responses have been processed. You may optionally provide your email address at the end if you would like to be contacted directly about results or follow-up studies.

**Instructions:**
Please answer the following questions in the context of a the model transformations **you have encountered**. In this way the survey attempts to capture opinions about TYPICAL transformations. Answers are meant to represent your own opinions and experiences. Below each question is a textbox for additional comments.

Example question:
"What effect does Small Transformation Size have on a transformation?"
If your experience is that the smaller transformations you have seen are *typically* more understandable, select "Increases" for Understandability.
If you believe smaller transformations are typically more understandable but only to a small degree, select "Somewhat increases".
If you believe it's typically not important for understandability, select "Has little effect".

FIGURE 7.2: Validation survey introductory text

## 7.4 Results and Processing

Fifteen respondents filled out the survey, including the original four developers interviewed during the exploratory study and the two remaining CARM developers. The rest were from the general public, including three from industry, five graduate-level students, as well as one of the four primary committers of the Eclipse QVTo implementation. Every respondent had significant experience in QVTo. Since a response is given for each quality goal/attribute pair, the results data set therefore contains 37 attributes * 4 goals * 15 responses = 2220 data points. In addition, 41 qualitative comments were added in the comment fields below each attribute.

The data were then processed. First, the CSV downloaded from Google Forms was converted to readable response sets, one per respondent, including comments (see Python script in Appendix B.1). In this format, questions where the respondent had commented that they did not understand the question being asked were manually marked as exempt.

Later in the analysis, these exempt answers are treated as missing values. In total, 15 responses were marked exempt over all respondents. For cases where there was ambiguity or a question asked in the comment, a follow-up was performed with the respondent for clarification via email.

Then, a depiction of the entire response set in the form of a colorful "ribbon" was produced (see Python script Appendix B.3). In this view, all responses are shown together with different colors representing each response option. By inspecting the ribbon, one gets a general overview of the nature of the responses. The script generates ribbons with responses ranging from red to green as well as ranging from orange to blue for better accessibility. The ribbon depiction of our survey results is presented in Figure 7.3.[1]

Finally, the Likert answers were encoded to the scale [-3,3] and the data were placed in a format more suitable for import into R. This encoding in particular allows R to understand the relative ordering of the response options. All additional analysis of the validation results was performed in R (see the Python script in Appendix B.2 and R code in Appendix B.4). The survey results are summarized in Table 7.1.

## 7.5 Analysis

From the colorful depictions in Figure 7.3, we see that the most popular answer was in fact "Has little/no effect". This is however not surprising since it was not suspected that every attribute relates to every quality goal. We also see that there are more positive answers than negative. This is also expected due to the consistent directionality chosen for the questions. We also see a number of quality goal/attribute pairs with very high agreement that the attribute is important for quality (i.e. all greens or oranges), such as "Detailed Comments Throughout Code"/Understandability, "High Test Coverage"/Functionality, and "Little Dead Code"/Maintainability and Understandability. We also see cases where respondents agree that the attribute actually *hurts* quality (i.e. mostly reds or blues). For example, "Little Overloading of Mappings" hurts Maintainability. There, by looking at the survey comments, we see that respondents actually consider overloading to be "one of QVTo's nice features, keeping mappings clean and understandable" and that they "prefer overloading above additional parameters or internal switch statements". We can also see that the questions marked exempt are clustered around specific attributes. For example, five respondents have "Inheritance usage matches a metamodel" exempt, leaving only ten responses for that attribute in our data set. These attributes in particular, should be investigated in future work.

In addition to their qualitative value, surveys lend themselves to statistical analysis. However, like with qualitative methods in general, it is extremely important that only appropriate statistical methods are used on a data set lest misleading information be conveyed. Most statistical techniques can be categorized as either *parametric* or *non-parametric*. Parametric techniques rely on the calculation of means and standard deviations, and are therefore appropriate for *interval data*, meaning that the data is numeric and that the differences in values are also meaningful (e.g. the distance from 0 to 1 is the same as the distance from 1 to 2). Since it is hard to guarantee with Likert-style questions that respondents consider the distance from "Neutral" to "Agree" to be the same as from

[1] This depiction includes only fourteen of the fifteen responses, since one was received after the depiction was created.


FIGURE 7.3: **Colorful depictions providing an overview of survey responses.** Each column represents the responses of one respondent. The black, vertical line separates CARM responses from public responses, where CARM responses are on the right. "NULL" (white color) signifies an exempt response. Two color palettes are displayed to increase accessibility and the key for each palette is displayed at the bottom left.

| Quality attribute[2] | Quality goals | *Validation* F | U | P | M |
|---|---|---|---|---|---|
| | QVTo | | | | |
| Deletion uses trashbin pattern | P+ | 0(0,0) | -1(-1,0) | 1.5(0,2.75) | 0(-.75,0) |
| Few blackboxes | U+, M+ | 0(-1,0) | 1(.25,2) | 0(-1,0) | 2(1,2) |
| Few configuration properties | U+, P- | 0(0,.75) | 0(0,1) | 0(0,0) | 0(0,1) |
| Few intermediate properties | U+ | -.5(-1,0) | 0(-1,.75) | 0(0,0) | 0(-.75,1) |
| Few `end` sections | U+ | 0(0,0) | 0(0,1) | 0(0,0) | 0(0,1) |
| Few queries with side-effects | U+ | 0(-.1,0) | 2(1,2) | 0(0,0) | 1(1,2) |
| Few `when` and `where` clauses | U+ | 0(-.75,0) | .5(0,1) | 0(-.75,.75) | 0(0,1) |
| Little imperative programming | U+, M+ | 0(-.5,0) | 1(0,1.5) | 0(-.5,.5) | 1(.5,1.5) |
| Mappings only used when tracing needed | U-, P+ | 0(0,0) | 0(-1,0) | ~~1(0,2)~~ | 0(-1,0) |
| Minimal reassignment of objects | F+ | 0(-.5,1) | 0(-1,2) | ~~1(0,1)~~ | 0(-1,2) |
| More mappings than helpers | U+, P- | 0(0,.5) | 0(0,1) | 0(0,0) | 0(0,.75) |
| More queries than helpers | U+, P+ | 0(0,0) | 0(0,1) | 0(0,0) | 0(0,1) |
| Small `init` sections | U+ | 0(0,0) | 1(0,1) | 0(0,0) | 1(0,1) |
| | MMT | | | | |
| Confluence satisfied | F+ | 0(0,1.75) | 0(-.75,0) | 0(0,0) | 0(0,1.75) |
| Few input/output models | U+, M+ | 0(-1,0) | 1(1,1.5) | 0(0,1) | 1(1,2) |
| Inheritance usage matches metamodel | U+ | .5(0,1) | .5(0,2) | 0(0,0) | 0(-.75,1.75) |
| Pre- and post-conditions specified | F+, M+ | 0(0,1) | 1(1,2) | 0(0,0) | 1(.5,2) |
| | GPL | | | | |
| Detailed comments throughout code | U+ | 0(0,.5) | 2(2,3) | 0(0,0) | 2(1,2.5) |
| Few dependencies between functions | U+, M+ | 0(0,0) | 1(0,1) | 0(0,0) | 1(0,1) |
| Few dependencies on other modules | U+, M+ | 0(0,.5) | 1(0,1.5) | 0(0,1) | 0(-2,2) |
| Few mapping arguments | P+, U+ | 0(0,0) | 1(-.5,2) | 0(0,1) | 0(-1,1.5) |
| Few nested `if` statements | U+ | 0(-.5,0) | 1(1,2) | 0(0,0) | 1(1,2) |
| Formatting conventions followed | U+ | 0(0,0) | 2(1.5,2) | 0(0,0) | 2(1,2) |
| High test coverage | F+, M+ | 2(0,2) | 0(0,.5) | 0(0,0) | 1(0,1) |
| High usage of design patterns | U+ | ~~1(0,1)~~ | 1(1,2) | 0(0,1) | 1(1,2) |
| Interdependent functions near each other | U+, M+ | 0(0,0) | 2(1,2) | 0(0,0) | 1(1,2) |
| Little dead code | U+, M+ | 0(0,0) | 2(1.5,2) | 0(0,1) | 2(1.5,2) |
| Little overloading | U+ | 0(-1,0) | -1(-1,1) | 0(0,.5) | -1(-1,1) |
| Low code duplication with other modules | M+ | 0(0,.5) | 1(1,2) | 0(0,0) | 2(1.5,2) |
| Low code duplication within module | U+, M+ | 0(0,0) | 1(1,2) | 0(0,0) | 2(2,2) |
| Low execution time | P+ | 0(0,0) | 0(0,0) | 3(2,3) | 0(0,0) |
| Low syntactic complexity | U+ | 0(-1,0) | 1(1,1) | 0(0,0) | 1(1,1) |
| Short function chains | U+ | 0(0,0) | 1(0,1.5) | 0(0,1) | 1(0,1.5) |
| Small function size | U+ | 0(-.5,.5) | 2(1.5,2) | 0(-1,0.5) | 2(1,2) |
| Small transformation size | U+ | 0(0,0) | 2(1,2) | 0(-1,1) | 1(1,2) |
| Small interfaces to other modules | U+, M+ | 0(0,0) | 1(.25,2) | 0(0,0) | 1.5(.25,2) |
| Termination checked | F+ | 0(0,2) | 0(0,0) | 0(0,0) | 0(0,0) |

TABLE 7.1: **QVTo quality model validation results.** Validation responses ranging from "Strongly decreases" to "Strongly increases" are encoded on the scale [-3,3] and shown in the format [median] ([25th percentile], [75th percentile]). Validated attributes are shaded grey. Quality goal/attribute pairs which did not pass the sensitivity analysis are crossed out, and those which would have been included are shown in boxes. See Section 7.5)

"Agree" to "Strongly agree", non-parametric techniques can be used instead, which rely for instance on medians and rankings rather than means [90]. Nonetheless, parametric techniques are still the the most commonly used in the analysis of Likert scales, since they can sometimes provide higher-power conclusions [91].

In our analysis, we utilize parametric and non-parametric techniques simultaneously. Moreover, because each technique has weaknesses, it is crucial to also report descriptive statistics, such as the number of respondents to answer with a certain response on a given question [90].

### 7.5.1 Overall agreement

To measure overall agreement, we calculated the parametric *intraclass coefficient for intersubject concordance* ($ICC$) [92] and it's non-parametric counterpart Kendall's *coefficient of concordance* ($W$) [93]. These statistics measure respondent agreement where 1 implies perfect agreement and 0 implies no agreement. The $ICC$ assumes that the correlations between pairs of observers are equal for all pairs, whereas the only assumption for Kendall's $W$ is that the measurements are at least ordinal in nature [93]. For our response set, $ICC = 0.30$ and $W = 0.32$.[3] These values, which have been interpreted by some researchers as low agreement [94], suggest that at least some attributes may have high disagreement, but the precise interpretation must be considered on an individual basis. Therefore, to obtain more insight into the agreement and disagreement over specific quality goal/attribute pairs, we investigate interquartile ranges of each pair. The interquartile range, a descriptive statistic, represents 50% of developers having a responses which neighbor each other on the Likert scale (i.e. the distance between the 25th and 75th percentiles). This range is also displayed numerically in Table 7.1. There, we find that 80% of pairs had an interquartile range of one or less, which we consider high agreement for most attributes.

The pair which was least agreed-upon (i.e. the largest interquartile range) was "Few dependencies on other modules"/Maintainability. By reading the survey comments, we find that while some developers consider dependencies bad for quality, others consider the alternative to be higher code duplication, in which case more dependencies are preferred. We notice that the least-agreed upon attributes tend to be either transformation-local or high-level architecture attributes, from the classification we provided in Table 6.2. The pairs most agreed upon (i.e. interquartile range of zero) were largely where answers were neutral, so respondents agreed that those attribute/goal pairs are unimportant for quality. Besides those, it was the presentation-related attributes which appeared to have highest agreement. The GPL-applicable attributes also had higher agreement in general than the MMT- or QVTo- specific attributes.

### 7.5.2 Central tendencies

We now investigate the central tendencies of the responses. Here we report medians, since that is most appropriate for ordinal data. First, we consider which attribute/goal pairs are perceived as having the strongest impact on quality (i.e. the highest median response). The pair with the highest median was "Low execution time"/Performance

[3] All statistics reported here are significant at the $p < .01$ level unless stated otherwise.

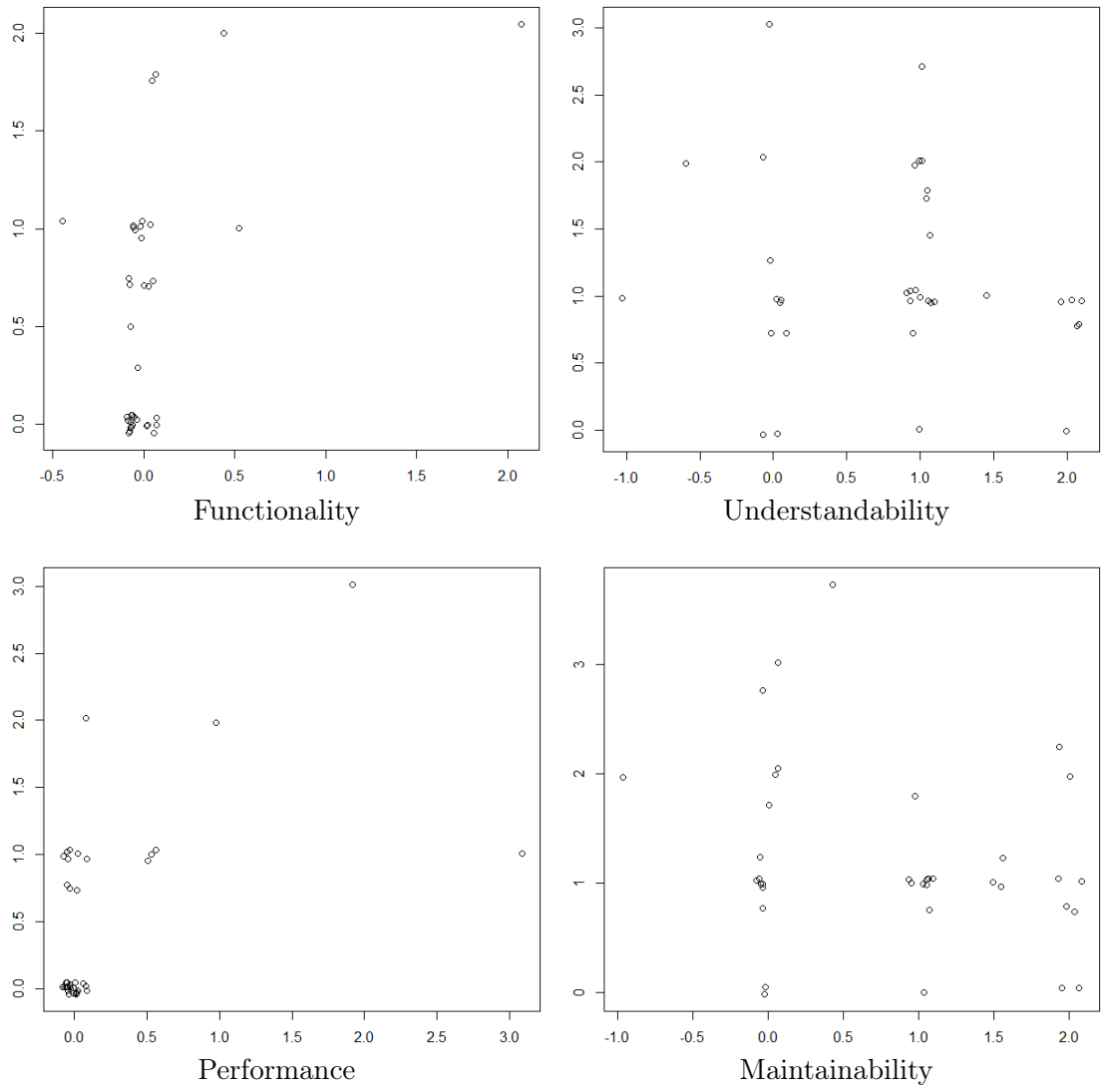


FIGURE 7.4: Plot showing medians versus interquartile ranges for each quality goal

followed by attributes "Little dead code", "High test coverage", and "Few blackboxes", among others. Four pairs yielded negative medians, indicating that developers perceive these to decrease quality rather than increase it. For one of these, "Deletion uses a trashbin pattern", a tradeoff is clearly acknowledged by the respondents where it decreases understandability but increases performance.

To consider the central tendencies together with agreement, we can display the pairs in a graph plotting medians versus interquartile ranges, as shown in Figure 7.4. There, we have split the data up according to quality goal, so we can also see the difference in responses for each goal. There, points towards the bottom represent high agreement and points towards the right represent positive responses. We again see that for functionality and performance, most attributes were agreed to have no effect, and that the plots for understandability and maintainability are very similar, likely due to the strong relationship between the quality goals. With this view, it is also easy to see outliers, which can then be found using Table 7.1. For functionality, for instance, we immediately see that "High test coverage" was agreed to have a strong, positive effect.

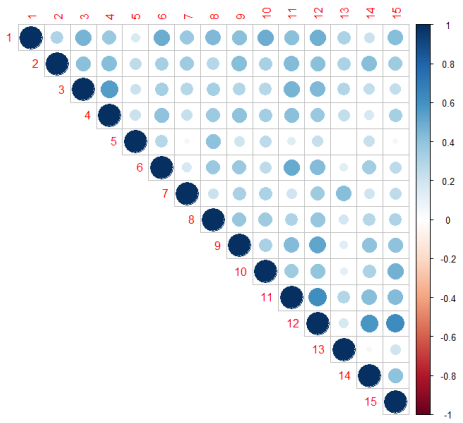


FIGURE 7.5: Pairwise correlations of responses between respondents

### 7.5.3 Differences between respondents

We also investigated response differences between four CARM developers interviewed and the two not interviewed. For those interviewed, $W = 0.61$, and for those not, $W = 0.56$, with similar ICC values. Therefore we do see slightly higher agreement amongst those interviewed, which could be due either to the fact that the attributes were built with their input, or because those developers have more experience in QVTo and therefore may have stronger feeling for certain attributes. We then looked at the difference in agreement between the CARM team and the public, interesting because it gives some insight into how generalizable the model is outside of their team. There, $W = 0.44$ for the CARM team and $W = 0.32$ for the public, so we do see in fact that there is more agreement within the CARM team than amongst members of the public. This is most likely because the general public do not have a common context in which they use QVTo, and therefore their opinions about the best way to write QVTo differ. We also notice that for members of the CARM team, the median answers for "Termination checked" and "Confluence satisfied" were considerably higher than for the public. This we explain by differences in academic background (two team members have PhDs in formal computer science topics, whereas members of the public did not have formal computer science backgrounds) as well as the CARM team's continuous contact with our researchers, potentially raising their awareness of these two concepts. Upon follow-up with the CARM team, however, each stated that they did know these concepts before our research began.

To see the relationship between individual survey respondent answers, we have plotted the pairwise correlations between responses in Figure 7.5. These correlations were calculated using the parametric statistic Spearman's $\rho$ , but the results were equivalent for the non-parametric Kendall's $\tau$. There, respondents 1 through 6 are members of the CARM team, and the rest are public. It is clear that some respondents had much more similar responses to some than others, for example respondent 12 correlated highly with 14 and 15. Looking deeper, we see that this high correlation comes primarily from GPL-applicable attributes, and that these three respondents in fact comprised those who reported having the least experience in QVTo. So, the correlation may be explained because these respondents are answering the survey questions based more on prior knowledge of code quality from GPLs than QVTo experience.

### 7.5.4 Validated attributes

Finally, we identify which attributes/goal pairs we actually consider validated by the survey. For this, we define two criteria which must be met by a pair. First, the median answer must be at least "Somewhat increases", therefore asserting that in general the attribute is thought to increase the quality goal. Second, at least 75% of the responses must have been at least "Has little/no effect", thereby requiring a base level of agreement amongst the respondents.

The results which satisfied these criteria are shaded gray in Table 7.1. Of our original 37 attributes, 26 are therefore validated for being important for at least one quality goal. Of those validated, nine were considered MMT- or QVTo-specific. Since a significant portion of our attributes were validated in this manner, we considered this a positive result.[4] Although we still do not rule out the importance of the other attributes, it is these validated attributes that should be considered most strongly for implementation in quality tooling.

### 7.5.5 Sensitivity analysis

To assess how sensitive our set of validated quality attribute/goal pairs is to the specific set of survey respondents, a sensitivity analysis was performed. Here, each respondent was removed one at a time and then the two validity criteria from Section 7.5.4 were rechecked. There were three pairs which during at least one of the rounds of the analysis did not satisfy the criteria. The three pairs which did not satisfy the criteria in every round are struck through in Table 7.1, two of which were QVTo-specific and one GPL-applicable. There are also four pairs which only satisfied the criteria when one respondent was removed. These pairs are displayed in boxes. So, additional investigation should be done for the struck-through and boxed attributes. It is also important to recall that some of the responses were marked exempt. For instance, "Inheritance usage matches a metamodel" is displayed in a box, but it only had ten responses included to begin with, so more validation is required in particular for those.

## 7.6 Threats to Validity

Our approach is not without limitations. First, the quality model is not a complete picture of QVTo quality, since it only contains the attributes representing the most important issues at the time of our research. Second, many of the attributes are also still heavily influenced by the way of working at ASML, despite the triangulation approach. If another team had been interviewed instead it is therefore possible that the quality model would contain different attributes. For example, one non-CARM developer who took the evaluation survey asked why no attributes asked about resolving, since using resolving when not necessary was something that developer felt lowered quality.

[4] While in the first author's opinion the more validated the better, that was not the opinion of everyone. For example, one of the advisor's of this project was disappointed to hear that so many were confirmed as important, since finding only a couple as important greatly eases the burden of making tools that address all quality concerns.

Third, a large portion of our approach is based on perceptions, which are inevitably biased. So, even though we build our quality model with no prior conception of what quality in QVTo should mean, developer perceptions can nonetheless be based on previous impressions of software quality. Fourth, the validation survey format could be improved. It was noted by some respondents that the strict question format makes some answers obvious while others feel oversimplified. These feelings could hurt response validity or cause others to opt not to take the survey. Lastly, small sample sizes in our interviews and evaluation makes our results susceptible to overfitting and sampling bias. This is, however, a common concession when adopting the pragmatist stance [17].

# Chapter 8

# Conclusions of Part I

In this part we addressed the research question, how can we assess quality of QVTo model transformations? To that end, we constructed a quality model for QVTo transformations. This model consisted of 37 quality attributes together with evaluation procedures, which corresponded to 6 high-level quality goals identified as being important for QVTo in practice. The attributes were evaluated using a survey of experts, after which 26 can be considered validated. The attributes that proved to be important to QVTo included not only those which have counterparts from GPLs, but also some specific to QVTo itself. This suggests that any quality model not made specifically for QVTo may leave out some important factors in assessing QVTo transformation quality. In addition to the QVTo quality model, a secondary contribution was the QVTo code best practices and difficulties presented in Section 5.1, which can be leveraged by QVTo practitioners to improve their QVTo code or by researchers to identify interesting areas for future work.

Furthermore, to build the quality model, a comprehensive exploratory study was performed in order to gather rich qualitative data. Our approach, including existing material review, expert interviews, and introspection, was driven by our pragmatist pragmatist philosophical stance, which puts usefulness within the actual practitioner context at the forefront.

The work presented in this part has also been submitted in a paper [19] to the Quality in Model Driven Engineering track of the 9th International Conference on the Quality of Information and Communications Technology (QUATIC '14) [20]. Included in the reviewer feedback was that additional bias may be present in current QVTo experts, since most current QVTo experts are tool experts; additional research in the context of other companies would be interesting; and it would be useful to elaborating on the seriousness of the threats to validity. The feedback in general however was very positive: "The paper has many more pros than cons" and "Despite the merit of the quality model itself, which is a very good contribution per-se, he paper also offers a good description of the process used to develop and validate the model, which is also a very valuable contribution to the community". Since the paper was chosen as one of the best within the track, we have also been invited to submit an extended version to the main track.

## 8.1 Future Work for the Quality Model

The quality model could be made more complete, i.e. areas should be identified where the attributes still do not fully address the quality goals. The attributes should then be minimized so that the attributes required to address each quality goal are kept to a minimum. Having a minimal set is important both to improve computation times of any supporting tools, but also to improve usability of the model by practitioners. A potential technique to perform the minimization is factor analysis [95] in order to remove redundant attributes. Additional evaluation of the quality model should also be performed. Evaluation techniques such as that from [42] could be used to provide additional support for some of the quality attributes presented here.

As it is, the quality model here can already be leveraged in future work by performing additional validation of the quality model and by developing tools which implement the evaluation procedures corresponding to each quality attribute. Implementing the evaluation procedures is namely the last step of the MDE framework presented in Section 3.3. Choosing tools to implement and an approach to designing their features are therefore the subject of Part II of this thesis.

# Part II

# Developing higher-quality transformations

# Chapter 9

# Approach

In Part I a quality model for QVTo transformations was constructed. The objective of Part II is to investigate our second research question, namely, how do we create higher-quality QVTo transformations? Specifically, since developers create the transformations, how can we use the QVTo quality model to help developers create higher-quality model transformations?

As per the seventh step in the MDE quality model framework introduced in Section 3.3.2, we must now *execute* the quality model by implementing the evaluation procedures of the quality attributes. Therefore in this part, we seek to create and evaluate a tool which implements one or more quality attributes.

The need for tooling is further stressed in literature. Kolahdouz-Rahimi et al. [50] notes that although the importance of metrics tools for transformations is still at an early stage, the need for tools is quickly increasing as transformations become more complex. Syriani and Gray [16] also identified the distinct lack of tooling for MDE as one of the current biggest challenges in developing high-quality model transformations. Moreover, according to the pragmatist stance, developing tooling based on the quality model to be used *in practice* serves as additional and extremely valuable validation of the quality model itself.

We discuss ideas for tools to develop based on the quality model, as well as the tool we have selected for implementation, in Section 9.1. We then describe our approach to developing tools in Section 9.2. In the remaining chapters of this part, we discuss the tool requirements (Chapter 10), tool implementation (Chapter 11), and tool evaluation and future work (Chapter 12).

## 9.1 Tool Ideas

The attributes from our quality model support many directions for tool development. Here, we identify a few, which we have grouped in informal categories. At the end of this section, we select one tool for implementation. The remaining tools identified here, however, are still considered promising areas for future work.

First there are tools which simply measure and report metrics. These could for instance measure metrics over time and/or across projects and could be implemented as either

standalone assessment tools or integrated developer tooling. The tool developed by Nguyen [69] described in Section 5.2.2 is an example of a standalone metrics tool for QVTo. In the case of our QVTo quality model, a tool could measure all attributes or just the best-validated ones.

Another category is visualization tools. Visualizations are often based on measured metrics, so like metric tools, they could for instance depict changes over time. They can also show structure rather than metric values. For example, the transformation analysis tool [74] described in Section 5.2.2 is a visualization tool to help developers reduce intramodule dependencies and increase development speed. Similar visual treatment could be provided for some of the high-level architecture or transformation-local metrics, including "Small interfaces to other modules", "Short function chains", and "Interdependent functions near each other". Although not well-validated by the expert survey, "Inheritance usage matches a metamodel" could also yield interesting, novel visualization tool opportunities. Van Amstel, van den Brand, and Serebrenik [96] also presented an approach for visualizing traceability in model transformations using TraceVis. So another tool could be integrating and adapting the approach presented there into a development toolset (in our case for the CARM team).

A third tool category is analysis tools. Such tools could check formal behavioral conditions such as "Termination checked" and "Confluence satisfied". Neither of those however were validated with our survey, so they make less attractive tool candidates. An analysis tool addressing a well-validated attribute would be a profiling tool, namely addressing attribute "Low execution time". This tool is also less attractive for implementation here, since the CARM team already utilizes a simple profiler tool that was developed in-house.

Finally, there are process-oriented tools which are best described as developer tooling. One such tool is a code coverage tool which measures and displays test coverage of the transformations. A code coverage tool addresses the "High test coverage attribute", which was validated to be important for both functionality and maintainability. A code coverage tool could also be developed in such a way that it also helps identify dead code, addressing another well-validated attribute from our quality model. Another candidate for developer tooling based on our quality model is refactoring support. For many GPLs, refactoring support is built into the IDE, for instance for renaming variables or refactoring code to use design patterns. Refactoring and design patterns, for GPLs as well as model transformations, has received much attention in research [16, 47, 60]. In our model, refactoring support could address "Formatting conventions followed", "Few queries with side-effects", the attributes concerning code duplication, "Deletion uses trashbin pattern", and "High usage of design patterns".

### 9.1.1 Chosen Tool

Of the options presented above, we have chosen to implement the test coverage tool. The "High test coverage" attribute was well-validated and moreover, the lack of a test coverage tool was mentioned by every developer during the expert interviews as an area requiring attention, as it is a standard part of the GPL developer toolset. Therefore, a test coverage tool seemed to provide the clearest benefit to the developers in improving the identified transformation quality goals. We chose to develop only one tool to maximize the time to improve the tool, rather than creating multiple mediocre tools. This is particularly important because good integration and usability is essential to the success of a tool [75],

and tools without enough attention paid there would likely be abandoned by developers. In our case, due to the pragmatist stance, the success is defined as success within a particular context. Therefore, we strive to create a tool which is useful to developers of the CARM team. Our tool evaluation approach is discussed in Section 9.2.

To increase the usefulness of the tool, we develop this tool so that it can indeed support both attributes "High test coverage" and "Little dead code". The requirements of the test coverage tool are formulated in Chapter 10.

## 9.2 Tool Development Approach

Hall and Fenton [44] provide a number of general recommendations for implementing successful metrics programs. Although our tool is not necessarily a "metrics" tool, most of the recommendations still apply. Among the recommendations are: incremental implementation, transparency of data collection, that the usefulness of the tool should be apparent to all practitioners, developer participation in defining metrics, practitioner confidence in the metric collection, automated data collection, and practitioner training. Paige and Varró [75] also noted in their experience developing MDE tooling for ten years, that in general rich graphical views of models are more useful than editing tools. These recommendations are taken into account when designing our tool.

Already leveraging two suggestions from [44], we follow an iterative development approach with significant developer involvement. This iterative approach is in fact form of action research, discussed in Section 7.1.2.2. Action research lends itself to the pragmatist stance, since every iteration incorporates developer feedback to maximize the tool's usefulness according to their context and needs.

We followed the following steps in our development process:

1. Develop a proof of concept exemplifying the possibilities for the quality attribute(s).
2. Present the proof of concept to the QVTo practitioners and get initial feedback.
3. Formalize the tool requirements and review them with practitioners.
4. Develop the prototype and provide it to the QVTo practitioners for normal use.
5. After a period of use, get feedback from developers and improve the tool and provide an updated version to developers.
6. After an additional period of use, perform a final evaluation of the tool with developers and identify areas for future work.

In addition to the basic tool functionality, we also collect *usage statistics*. These statistics are primarily intended to help with tool evaluation. For ethical reasons, developers are made fully aware of the type of data being collected. How the usage data collection is implemented and the impact it may have on developer behavior is discussed in Section 11.4.4.

The tool developed has also been made *open source*, so that it can be downloaded and used freely by the public. Open-sourcing greatly increases the impact of the tool developed here. More details on the open sourcing of the tool are provided in Section 11.4.7.

## 9.3 Quality Tool Evaluation

To answer our research question, as well as complete the seventh step of the MDE framework, the tool must be evaluated in its ability to help developers create higher-quality transformations. Because our development approach utilized action research, some initial validity assumed for the tool. However, additional evaluation is desirable. Kitchenham, Linkman, and Law [97] present a methodology for evaluating software engineering tools. There, they identified nine methods along with criteria to help evaluators select an appropriate method to evaluate their software. In particular, the methodology can be used by tool vendors seeking to demonstrate advantages of a product or researchers developing new software methods. The authors grouped evaluation methods in two main categories: quantitative methods which establish measurable effects of using a tool, and qualitative methods which establish the appropriateness of a tool in a giving setting, typically by gathering opinions from practitioners on tool features.

Three of the methods fall into the quantitative category: quantitative experiments, quantitative case studies, and quantitative surveys. These are appropriate when the benefits of a tool are readily quantifiable. The remaining methods are qualitative: screening, where an individual performs a feature evaluation using literature rather than the tool itself; qualitative experiments, where a group performs a feature evaluation with typical tasks in mind; qualitative case study, where the tool is used on a project; qualitative survey, like the experiment but with voluntary participation; and two hybrid methods (a.k.a. mixed methods). The authors give recommendations for when to use each method based on the time, people, and information available to the evaluators. These recommendations are leveraged in choosing the evaluation method for the tool.

### 9.3.1 Our Evaluation Approach

Based on our pragmatist stance, we consider the tool successful if it is useful for developers in their specific context in improving transformation quality. Given our available resources and pragmatist stance, we therefore evaluate the tool through a combination of three methods (in fact a form of mixed-methods research, one of the techniques most suitable for the pragmatist stance, as discussed in Section 1.1).

First, because we have direct access to practitioners over a period of time, we leverage a qualitative case study as an evaluation method. Specifically, the CARM team uses the tool for a period of time on their running QVTo projects and then practitioner feedback is gathered at the end. Practitioner feedback is essential for the pragmatist stance because practitioners are the only ones who can actually judge how useful the tool is in practice. This feedback is gathered through semi-structured group interviews. Group interviews were chosen because it allows the researcher to easily follow up on suggestions by the developers and also allows the developers to discuss ideas with each other to reach consensus on the spot.

Second, since we are collecting usage data, we include an analysis of usage statistics as a second evaluation method. This is a form of quantitative case study, since we can easily quantify some of the statistics collected over the usage period. The usage data also complements the qualitative analysis, since we can use it to quality the developer's feedback. For example, very positive feedback with a record of heavy usage is much more convincing than positive feedback where the tool was barely used.

Finally, we perform an additional quantitative case study in the form of a timing analysis to assess the tool's performance. Performance is an important factor in the tool's usability, therefore directly affecting its usefulness to developers in a practical setting. This analysis does not require prolonged access to the CARM developers, allowing it to be performed without taking up more time of the CARM developers. Since this analysis provides additional data with which to cross-reference developer feedback, and allows other practitioners not participating in our case studies to assess how useful the tool may be in their own context, the timing analysis complements the other evaluation methods used.

# Chapter 10

# Coverage Tool Requirements

As discussed in Chapter 9, it was selected to implement a code coverage tool for QVTo. First we describe the specific context for which the tool will be implemented Section 10.1. We present work related to code coverage tooling in Section 10.2. Finally we formulate the requirements for our QVTo code coverage tool in Section 10.3. We use the terms *code coverage* and *test coverage* are used interchangeably, since both are commonly used in industry to refer to test coverage tools.

## 10.1 Technical Context

It is essential that a tool is appropriate for the specific technical environment in which it will be used. Therefore, before identifying specific requirements, we review the current testing procedures used by the CARM team, in which this tool will first be used. We also consider the generalizability of this environment in order to assess how well a tool developed for the CARM team could be ported to work with other teams. According to the pragmatist stance, however, optimizing for the CARM development context is first priority.

As mentioned in Section 2.3, the CARM team tests their transformations using unit tests. There, test cases are created which specify one or more input models, perform the specified transformation, and compare the transformation output model(s) to some previously-created output model. If the models match, the test case passes, and if not it fails. A depiction of this process of executing a single test case is given in Figure 10.1. According to Rahim and Whittle's [50] survey of model transformation verification and Lin, Zhang, and Gray's [4] description of a testing framework for model transformation, this is a common setup for testing MMTs. The maintainers of the Eclipse QVTo implementation also use the same testing approach. Therefore, a tool developed for this setup is likely to also be applicable to other teams.

The unit tests written by the CARM team, like the tests written by the Eclipse QVTo maintainers, leverages the JUnit testing framework, in which the transformation being tested is called through Java. Output and expected models are compared using EMF-Compare [98]. Test cases are organized in test sets. Generally one test set is created per transformation in the CARM team. Test sets are then combined into a test suite, which runs all test sets. The test runs are started from within Eclipse using the JUnit launch

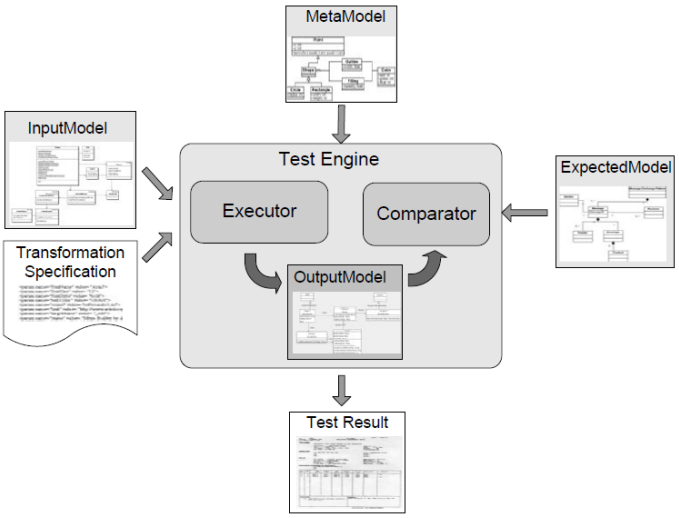


FIGURE 10.1: Model transformation unit test case execution [4]

configuration, where it can be specified whether to run a suite, set, or individual test case. Parameters necessary for the launch are set within the JUnit launch configuration.

## 10.2 Related Work

Rahim and Whittle surveyed current work in model transformation verification [50], where they use the term verification to mean from formal analysis to quality assurance. The majority of work is focused on generating test data (e.g. input and expected models) and on model comparison (i.e. comparing output models to the expected model). However, since we are building a tool exclusively for code coverage and because the CARM team already has a test framework, test generation and model comparison are not in the scope of our tool. The authors also provide eight important areas for future research, three of which we mention here. The first is a distinct lack of end-to-end verification tooling, which we address by providing a coverage tool fully integrated into the testing environment already used by the developers. The second area for research is grounding future work in industrial practice, since the vast majority of MMT verification research is very academic and not immediately applicable in industry. Here, the development of the tool was grounded heavily in industrial practice as a result of the pragmatist stance. A final area recommended for future research is attaining transformation language independent forms of verification since industry makes use of many different transformation languages. We do not follow this recommendation, however, since according to the pragmatist stance we should optimize the tool for its context, namely leveraging QVTo-specific attributes when useful to do so. Moreover, this recommendation is at odds with the recommendation for end-to-end tooling, since the environment in which the tool must be integrated is also language-dependent.

A number of coverage criteria have also been proposed for MMTs. These criteria can be divided in three categories: input metamodel coverage, transformation coverage, and generated code coverage [72]. Metamodel coverage, a form of black-box testing since it does not require access to transformation source code, measures how many elements from the metamodel are used by a transformation. A challenge with metamodel coverage

is that metamodels are generally large, therefore requiring an enormous number of tests to cover the entire metamodel. Even with facilities to generate these tests, test suites may still require a large amount of time to run. Moreover, many transformations are only intended to address a small part of a metamodel, in which case very low measurements of metamodel coverage are expected and acceptable. Transformation coverage, on the other hand, measures how much of the transformation source code is covered. Coverage variations for transformations include rule coverage, instruction coverage, and (for MMT languages supporting conditions) decision coverage. Generated code coverage, only applicable to transformations which generate code, measures traditional code coverage criteria on the generated code. Traditional coverage criteria include statement coverage, function coverage, and class coverage [72].

Marick [99], although agreeing that a measurement of code coverage is useful, describes how developers tend to *misuse* code coverage measurements. In particular, code coverage cannot detect faults *of omission*, where the lack of code or functionality is not tested. For example, a code coverage statistic won't tell a developer if there is too little error handling. Instead, a code coverage tool makes developers want to write additional "quick and dirty" tests just to increase the coverage metric. To correct for this, the author provides two recommendations: First, do not use code coverage metrics during initial test design. Second, do not require 100% code coverage. We keep these in mind while designing our tool.

## 10.3 Requirements

To start we identify two initial assumptions, based on the technical context described in Section 10.1: First, JUnit test cases are present for the transformation. Second, these tests cases are run within Eclipse using JUnit launch configurations. The goal of the coverage tool developed here is therefore to add test coverage functionality to those test executions.

The requirements gathering process, reflecting our development process, was iterative. Although we present our implementation in Chapter 11, we present the evolution of requirements throughout the project here.

### 10.3.1 Initial Requirements

First, according to our development process described in Section 9.2, a proof of concept was developed to exemplify some possible tool features. Our proof of concept demonstrated two features: the ability to add highlighting to code in a file in Eclipse, and the ability to count the number of mappings touched during a single transformation execution. These features were chosen since they represent core features of code coverage tools for GPLs (e.g. EclEmma [100]). The proof of concept was implemented as a small Eclipse plugin and could be demoed by using the QVTo launch configuration in Eclipse.

Per step two in our process, this proof of concept was demonstrated to developers to get feedback. From this feedback, the following list of initial requirements was formulated, step three of our development process. Many of the requirements are based on features of GPL code coverage tools which developers have come to expect in a code coverage tool.

- REQ1. Coverage should be able to be collected on multiple test cases at once, i.e. when a test set or test suite is run.
- REQ2. Coverage should be calculated for imported libraries as well as transformations.
- REQ3. Code should be highlighted in the editor, green for visited and red for unvisited, ideally with fine granularity (e.g. line-based).
- REQ4. Percentages representing the calculated coverage based on the desired coverage criteria should be displayed in a new view inside Eclipse. (The specific coverage criteria are discussed in Section 10.3.2.)
- REQ5. Aggregated coverage statistics should be displayed for a project and then able to be drilled down to individual modules.
- REQ6. The tool should make use of existing JUnit launch configurations in order to avoid duplicating the settings needed to launch the test.

From these requirements, a prototype was developed per the fourth step of our development process. This prototype satisfied all initial requirements except REQ3. REQ3 was not fulfilled because while obtaining the visited expressions was possible in a similar way to mappings (details of Implementation are described in Chapter 11), obtaining the unvisited expressions proved complicated due to how the language is implemented. Therefore REQ3 was only partially satisfied in the prototype: visited expressions were highlighted green, but red highlighting was only added to unvisited mappings, helpers, and queries. Unvisited expressions within a function that was visited therefore received no highlighting.

The prototype was used by developers for three weeks. Developer feedback was then gathered through an informal group interview per step five of our process. Developers were overall very satisfied with the tool, stating that even with current features, they were "impressed" and that the plugin "solved their needs" for code coverage. They also had many questions about the implementation details, since the team had run into difficulties in creating similar tooling (for example for profiling) in the past, which thanks to significant contact with the QVTo community we were able to solve in the code coverage tool. Some of the specific challenges overcome are described in Chapter 11. The developers also suggested some additional features to be added to the tool, which have been formalized in the additional requirement below:

- REQ7. Draw attention to transformations with very high or low coverage by coloring them inside the view that displays percentages. Percentages below a certain threshold should be colored red and those above a certain threshold should be green. Ideally these thresholds are settable by the tool users.

This requirement also addresses the second recommendation made by [99] mentioned in Section 10.2 by allowing developers to explicitly lower the high threshold for coverage (elaborated in Section 11.4.5), therefore not defining acceptable coverage as 100%. The first recommendation, namely not to use coverage analysis during test design, has been addressed by adding a clause to the instructions of the tool suggesting developers take code coverage metrics into account only after the tests have been designed.

Furthermore, the developers expressed that it would still be best if REQ3 could be fully implemented, highlighting unvisited expressions in red. Therefore, REQ3 remained as an unchanged requirement.

### 10.3.2 Coverage Criteria

Of the three coverage criteria categories described in Section 10.2, in our tool we focus on transformation coverage. This is because the test infrastructure used by the CARM team (and likely by other teams) are written to test certain parts of a transformation, just like testing normal code, making transformation coverage the metric most directly useful to developers in assessing their test coverage.

To our knowledge, no coverage criteria have been identified by related work specifically for QVTo. Criteria specific to QVTo are necessary because QVTo offers language constructs that are not available in other languages. Notably, this violates the recommendation mentioned in Section 10.2 that MMT verification methods be language independent. The following coverage criteria were identified initially: mapping coverage, helper coverage, query coverage, and statement coverage (where statements could be defined as anything ending in a semicolon).

In the prototype, all criteria were calculated except query coverage and statement coverage. Statement coverage was not implemented because the abstract syntax of QVTo does not actually include the concept of a statement. Instead, it includes only the concept of *expressions*. Therefore, statement coverage was replaced with expression coverage as a coverage criterion. Query coverage was not implemented because in the QVTo abstract syntax in the Eclipse implementation, queries are not distinguished from helpers. While receiving feedback from the developers, it also came to light that in some projects there is heavy use of constructors in addition to the other function types and the developers also wanted to know which of those were being used. So, constructor coverage was added as a coverage criterion. The developers also mentioned that in addition to the coverage criteria already identified, it would be useful to have a total function coverage measurement as well, where mapping, helper, and constructor coverage are aggregated, in order to get a better overview of the coverage of a module. This is important in particular for the CARM code because many of their transformations and libraries have varying mixes of function types, so being able to compare coverage estimates easily requires some aggregate measure. Therefore, the final coverage criteria identified as useful in QVTo are:

- Mapping coverage
- Helper coverage
- Constructor coverage
- Total function coverage
- Expression coverage

For expression coverage, it is essential to point out that expressions are nested. Therefore, the percentage representing the total number of visited expressions divided by the total

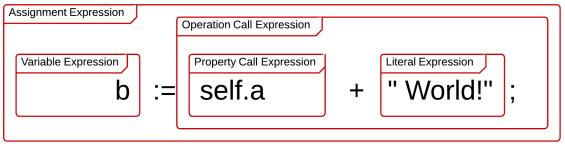


FIGURE 10.2: Nested structure of QVTo expressions shown to three levels deep

```
main() {
    source.rootObjects()[Root]->map Root2Root();
}

mapping Root :: Root2Root() : Root {
    element += self.allSubobjects()[A]
            ->select(a = "aoeu")->map A2B();
}

mapping A :: A2B() : B {
    result.id := self.id;
    b = new B(self);
}
```

(a) Leaf expression coverage

```
main() {
    source.rootObjects()[Root]->map Root2Root();
}

mapping Root :: Root2Root() : Root {
    element += self.allSubobjects()[A]
            ->select(a = "aoeu")->map A2B();
}

mapping A :: A2B() : B {
    result.id := self.id;
    b = new B(self);
}
```

(b) Total expression coverage

FIGURE 10.3: Expression coverage visualization options

number of unvisited expressions may not seem accurate. To illustrate the nested nature of expressions, a sample expression is given in Figure 10.2. In the figure a total of five expressions are shown. If the word " World!" were not not evaluated by the interpreter during execution (albeit not possible with this specific example), this code would yield an expression coverage of 4/5. Since it's hard for everyday users to see how that specific number was calculated without knowing the exact expression structure, this could be unintuitive. Therefore, two options were investigated for calculating and showing expression coverage: one option using all expressions, and one using coverage calculated using only with expressions, i.e. expressions with no subexpressions. To see the effect this would have, we compared the expression coverage percentages on a number of test sets. In fact, largely because there are so many expressions in a transformation anyway, the percentages reported by the two methods were actually quite similar. However, the two methods produce drastically different expression coverage visuals (required by REQ3). The difference is shown in Figure 10.3. For reference, the percentages for those examples were again similar[1]: the leaf expression coverage in Figure 10.3 was calculated as 6 out of 10 expressions = 60%, whereas the total expression coverage was 16 out of 29 expressions = 57%.

To determine which expression coverage measure to choose, according to the pragmatist stance, we must determine which is most useful. Therefore we demonstrated both options to the CARM developers using multiple test sets from the CARM code, showing both percentages and visuals. There, developers showed a very strong preference for the total expression coverage. This was because although one can deduce whether unhighlighted code is touched or not from the leaf expression coverage, it requires a great deal of effort from the developer. The developers even called the leaf coverage visual "barely usable" and noting that for them the visualization was probably the most important features of the tool. Good visualizations are also essential to address the quality attribute for "Little dead code", since it makes dead code make it much easier to identify untouched spots in the code. The developers were also asked whether they would prefer the visuals

[1]Two functions are not shown in Figure 10.3

using total expression coverage but percentage calculations reported using the leaf coverage. There, the developers felt that the percentages reported were equivalent for their purposes. In Figure 10.3 for instance in both cases it would be interpreted that almost two thirds of the code was tested, which would be compared to the coverage of other transformations in order to choose which transformation is most in need of additional test cases. Therefore, total expression coverage was used for the expression coverage criterion as well as in the expression coverage visual.

# Chapter 11

# Coverage Tool Implementation

Here we describe specifics of the coverage tool implementation. In Section 11.1 we present some preliminary information required to understand the tool architecture. In Section 11.2 we present the high-level and frontend architecture. To support our implementation and future tools, a number of patches were submitted to the QVTo engine to fix bugs and add new functionality, which we present in Section 11.3. In Section 11.4 we describe a number of additional key design decisions. Instructions for how one can build the plugins in order to add new features herself are provided in Appendix C.

## 11.1 Implementation Preliminaries

### 11.1.1 JUnit

To understand the coverage tool implementation even from a high level, it is important to understand how JUnit works inside Eclipse. To run a test, one creates a new JUnit launch configuration instance specifying the test to run. When this configuration is launched, the JUnit Eclipse plugin spawns a new Java process (therefore a new Java Virtual Machine, or VM) in which the tests are run. When the process finishes, the coverage data is displayed in the JUnit view inside Eclipse.

### 11.1.2 QVTo Engine

It is also essential to know the basics of how the engine interprets QVTo transformations. A transformation is executed by creating a new `InternalTransformationExecutor` class and providing it the transformation file and an input model. The CARM team has extended this class in their own code in order to provide additional functionality, such as registering additional metamodels before the transformation is executed. The executor then compiles the transformation and creates a `InternalEvaluator` class (which we also refer to as a *visitor*) which contains the code that knows how to handle each element type of QVTo's abstract syntax. For instance, the visitor contains methods `visitMappingOperation()` and `visitAssignExp()` to handle mappings and assignments, respectively. Coverage data for our tool can, then, be collected by *instrumenting* this visitor.

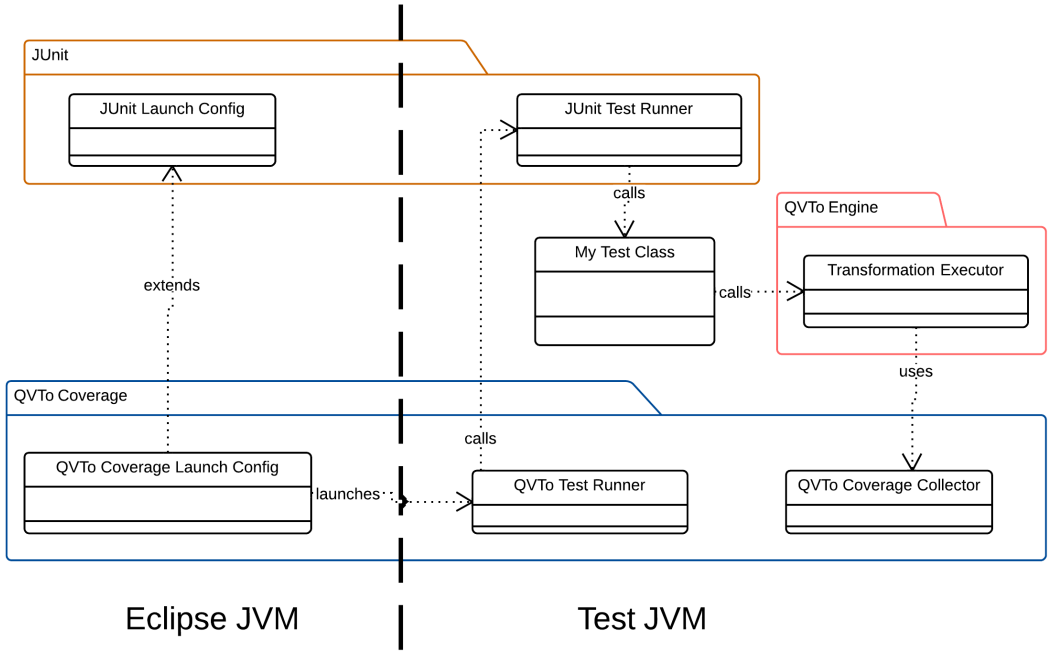


FIGURE 11.1: High-level architecture of QVTo Coverage Plugin

## 11.2 Tool Architecture

The QVTo code coverage tool developed in this research is implemented as plugins for Eclipse, to be used with the Eclipse QVTo implementation, as described in Chapter 10. The plugins work with Eclipse Kepler, the current version, and should work at least through the two next yearly Eclipse releases, Luna and Mars.[1]

### 11.2.1 High-level architecture

A high-level depiction of the architecture is provided in Figure 11.1. Note that this is an abstraction, so the entities there do not correspond one-to-one with Java classes. To satisfy REQ6, specifying that the tool should make use of existing JUnit launch configurations, the coverage tool contributes a new Eclipse launch configuration called the *QVTo coverage launch configuration* using the JUnit launch configuration extension point available from Eclipse. The QVTo coverage launch configuration extends the JUnit configuration, implementing the same run modes ("run" and "debug"), so that existing JUnit launch configurations can be run either as normal JUnit launches or as QVTo coverage launches. This implementation also satisfies REQ1, since directly reusing the existing launch configurations automatically provides support for running any test combinations and using the same environments that JUnit does.

When the QVTo launch configuration is invoked, instead of launching the JUnit test runner directly, the *QVTo test runner* is invoked. In the QVTo test runner a number of global variables are set and other setup is performed. Most importantly, an option is set inside the QVTo engine specifying that the QVTo interpreter should use the *QVTo*

[1] But as with most open source projects, it's hard to make guarantees.

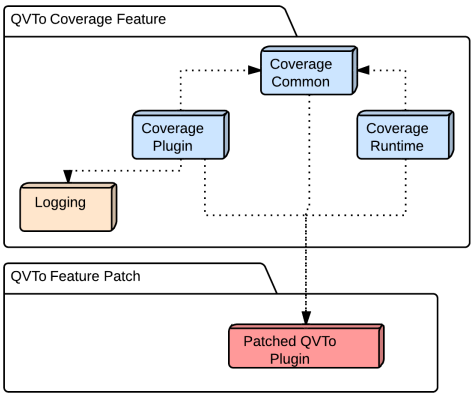


FIGURE 11.2: Individual plugins included in the QVTo Coverage Feature

*coverage collector* while the transformation is interpreted. This option is new in the QVTo engine, added specifically by us to support the coverage tool. We explain it together with the other patches contributed to the QVTo core in Section 11.3.

The setup code inside the QVTo test runner could not be performed inside the launch configuration class itself because the setup is only possible from within the same Java VM as the transformation execution. Therefore the setup must be done in a class after the new process has been spawned, making a shell class like our QVTo test runner an ideal solution. After necessary setup, the QVTo test runner launches the main method of the JUnit test runner, which runs the tests as normal. During interpretation of the transformation, however, the QVTo engine now invokes the coverage collector for every element visited in the transformation. The coverage collector maintains a list of all touched transformation elements. After the test run, this information is persisted to disk. When the JUnit test process finishes executing, control returns to the Eclipse VM and the QVTo coverage plugin reads the coverage data from disk and displays the coverage view to the user.

The QVTo coverage functionality is divided into a number of distinct Eclipse plugins. These plugins are then grouped into a Eclipse logical entity called a *feature*. Features allow one to easily manage dependencies and updates in Eclipse by adding them to *update sites*. A depiction of the plugins created for the coverage tool is presented in Figure 11.2. The Coverage plugin primarily contains classes required for the functionality on the Eclipse VM, the Runtime plugin contains classes needed in during interpretation on the Test VM, and the Common plugin contains classes required by both plugins. An additional plugin was created to handle logging, which is used for now only to record usage data (described more in Section 11.4). A feature containing the necessary patches to the QVTo engine is also a dependency of the coverage plugins, required for developers using the previous Eclipse version that wish to use the coverage tool. Once the next Eclipse version is released in late June 2014, this feature can be removed, since all required patches have been accepted into the next release.

As mentioned, the high-level architecture from Figure 11.1 is a simplification. As a significant effort was made to write the plugins in accordance with Java and object-oriented programming best practices, there are in fact 18 Java classes included in our implementation (not including QVTo patches) organized in 6 packages, totaling approximately

1700 lines of code. Instructions to browse the source code of the plugin are available in Section 11.4.7.

### 11.2.2 Frontend Architecture

The tool user is exposed to the frontend architecture of the tool, namely the views inside Eclipse. The frontend was designed to satisfy the remaining requirements. A screenshot of the tool after a test set has been run is given in Figure 11.3. On the bottom the QVTo Coverage View is shown. The left column of the coverage view shows a hierarchical view of transformations that were invoked during the test run, with project on top and the transformations within the project underneath. This satisfies REQ5 by showing coverage data both at the project (top) level and the module level. Both transformations and libraries are displayed under the project, satisfying requirement REQ2. The remaining columns show the coverage criteria, per REQ4. The coverage view cells are also colored according to the percentage calculated for that criterion. The thresholds are by default set at 30% and 90%, so coverage values less than 30% are shaded red and coverage values greater than 90% are shaded green, satisfying REQ7. The concrete values used to calculate the coverage percentages are also displayed in each cell to provide additional context to the user.

When the user double-clicks on the module name, the file is opened in the regular Eclipse editor view with the coverage overlay, addressing REQ3. The coverage overlay uses custom Eclipse *text markers* displayed over the visited and unvisited expressions to achieve the desired coloring. From the text markers tool users can easily see which code is not being touched by the test cases, for example the code after the `if` statement in the `A2B` mapping in Figure 11.3. Small green and red markers also show up right hand side of the editor, which provide users with an overview of the highlighting throughout the file, a particularly useful feature for large transformations.

## 11.3 QVTo Core Patches

A total of three patches were submitted and accepted into the Eclipse QVTo core engine over the course of this research. Two patches added new functionality and one patch fixed a bug in the engine. These patches together make it possible for not only our coverage tool, but also future tools such as an integrated profiler, to easily instrument the QVTo interpreter. These patches have already been committed by the Eclipse QVTo maintainers and will be released with Eclipse Luna on June 25, 2014 and development versions are already available for download. Therefore these patches serve as another contribution of this research. More information can be found on the patches using the provided links to the bug tracker issue and to the commit itself. In the current tool implementation made for Eclipse Kepler, these patches are added within a patch feature, as described is Section 11.2.1.

To submit a patch, the following steps must be taken:

1. Check out Eclipse QVTo implementation latest source code [101].
2. Make desired changes to code.

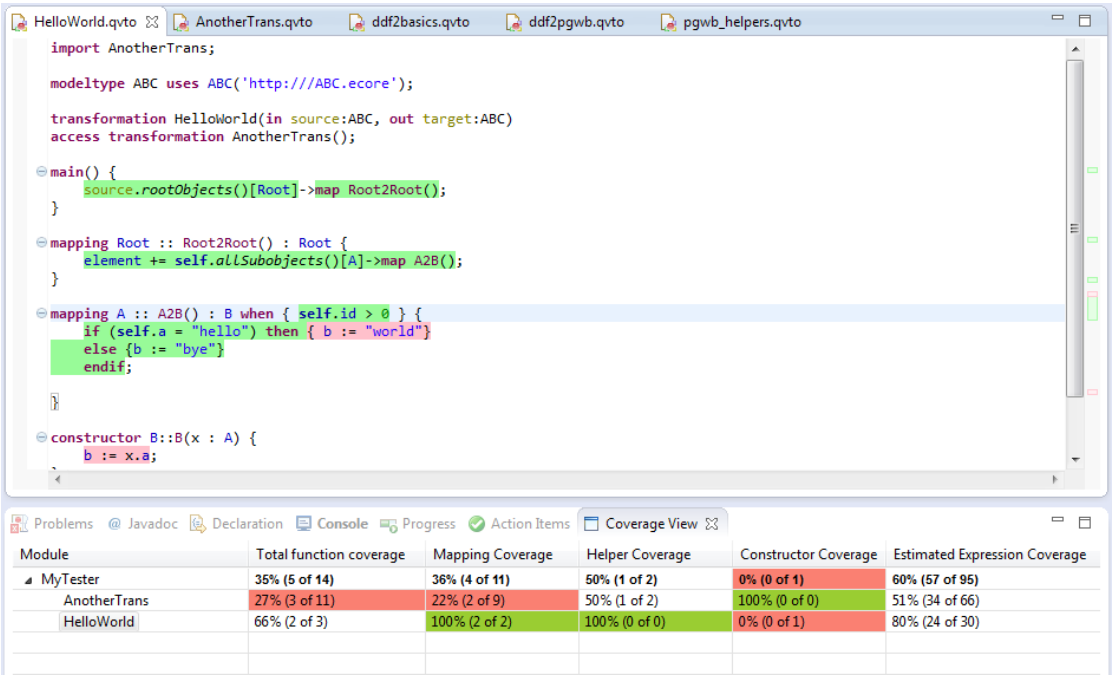


FIGURE 11.3: Coverage and editor views after a test run

3. Raise a new issue in the Eclipse bug tracker [102] describing the problem.

4. Generate a patch from the code changes and attach it to the bug report.

5. Update the patch based on QVTo maintainer feedback on the issue until the QVTo maintainers feel it is acceptable to commit. The QVTo maintainers will then commit the patch to the Eclipse release considered most fitting, for instance the next minor release.

### 11.3.1 Adding a visitor decorator class

Bug: https://bugs.eclipse.org/bugs/show_bug.cgi?id=430677
Commit: https://github.com/eclipse/qvto/commit/51028ae23d78e9d2b7832321254487458d8e3da7

A *decorator* is a software design pattern intended to enable attaching additional responsibilities to an object dynamically and transparently, without affecting other objects [5]. As opposed to inheritance, arbitrary combinations of decorators can be provided for an object dynamically, the result being that the additional functionality provided by each decorator are all incorporated into the final object. The decorator pattern is presented in Figure 11.4.

In the QVTo core, a simple decorator class was present containing the decorator methods for all visit methods for a visitor. This class was used for instance in the implementation of the built-in QVTo debugger. However, this class did not implement all interfaces required to decorate the `InternalEvaluator` class used to interpret QVTo transformations. Furthermore, the existing decorator required access to protected packages, meaning it could only be extended successfully by classes within the same package, making it impossible to contribute a 3rd-party decorator using that class. Therefore, we contributed a new decorator `QvtGenericVisitorDecorator` which extended the existing

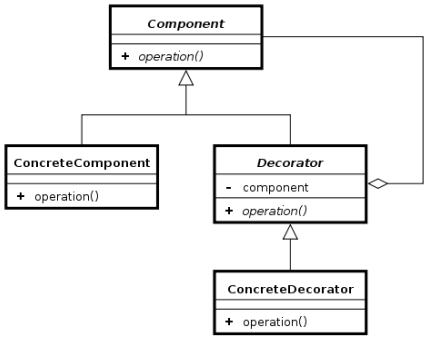


FIGURE 11.4: The decorator design pattern [5]

decorator and implements all required interfaces. Because the new class resides inside the QVTo core packages, default implementations were provided for the methods that required protected package access. In the terms of Figure 11.4, the Component is the `InternalEvaluator` class, the ConcreteComponent is the visitor class, and the Decorator is the new `QvtGenericVisitorDecorator` class. Third parties decorators can then easily extend `QvtGenericVisitorDecorator` to obtain a concrete decorator compatible with the QVTo visitor.

This patch also contributed a refactoring of the debugger implementation, leveraging the new decorator class. By using this new class, the QVTo debugger class became considerably easier to understand and saved approximately 50 lines of code in its implementation.

### 11.3.2 Fixing visitation of imported transformations

Bug: https://bugs.eclipse.org/bugs/show_bug.cgi?id=432969
Commit: https://github.com/eclipse/qvto/commit/d1aa7b9f5ca4c35d36f70031c889b7feec997ed7

During development of the coverage plugin, a bug was discovered in how the visitor used to visit imported transformations was created. Transformations can be imported for instance using the **`transformation access`** keywords. Specifically, while the visitor for typical transformations is created using a factory method of the `QvtOperationalEnvFactory` class, the "nested" visitor created for imported transformations was created always using the concrete (undecorated) visitor class. This meant that even if special settings were provided to the factory to create a special type of visitor, these settings would not be taken into account for nested visitors. This resulted in coverage data not being collected for imported transformations. Besides coverage data, this bug made it for instance impossible to use debugging breakpoints inside imported transformations, since the special debugger visitor was not being created. This patch therefore altered the method creating nested visitors so that it uses the `QvtOperationalEnvFactory` class to create the visitor rather than constructing the visitor directly.

### 11.3.3 Adding hooks for third-party decorators

Bug: https://bugs.eclipse.org/bugs/show_bug.cgi?id=432969
Commit: https://github.com/eclipse/qvto/commit/8160dd9f29509d7051e4961b36eeaea61fe7a377

This commit provides a method for third parties to contribute decorators for the QVTo visitor and have them be automatically incorporated during execution. Without this patch, it would not be possible to actually use a third party decorator without substantial modification to one's test cases.[2] The hook is created first by contributing a new `VISITOR_DECORATORS` option inside the `QVTEvaluationOptions` class. This option, by default empty, stores a list of visitor decorators extending the decorator class submitted in the patch described in Section 11.3.1. This list is then traversed by the `QvtOperationalEnvFactory` whenever it creates a new visitor. Therefore a QVTo visitor decorator can be transparently incorporated by adding the classname to the option list. Using options is also ideal because they can be set and overridden in a hierarchical manner, either globally or specific to the QVTo environment currently being used. This locality was in fact a requirement of the QVTo maintainers in order for this patch to be accepted. A unit test verifying the option's ability to add a decorator was also contributed to the QVTo test suite with this patch.

## 11.4 Key Design Decisions

A number of additional design decisions were made to support the coverage tool functionality and improve its usability. We highlight a selection here.

### 11.4.1 Inter-VM communication

One major decision, and also previously a barrier for the CARM team in implementing similar tools, was how to perform the communication between the Eclipse VM and the Test VM. As mentioned in Section 11.2.1, this happens in the coverage tool implemented here by writing the coverage data to disk as the test executes and then reading it back once the test execution has finished in order to display it to the user.

Specifically, there are three classes inside the Common plugin which store coverage data collection: `CoverageData`, `TransformationCoverageData`, and `NodeCoverageData`. These classes store coverage data for test runs, individual transformations, and individual abstract syntax nodes, respectively. Each of these classes implements the Java `Serializable` class so that the objects can be directly written and read from disk. During execution, only data on transformations and nodes that *are* visited is stored. The data is stored using HashSets inside each Data object to guarantee visited objects are only stored in the list once (since a transformation execution often visits the same expression multiple times). The unvisited data is calculated only after the test run has finished in order to reduce overhead during test execution and reduce the amount of data written to disk.

After execution, the Data objects are read from disk by the Coverage plugin. The objects are then used to populate instances of three new classes: `ProjectCoverageModel`,

[2] To use a decorator without this patch, one would need to extend the `InternalTransformationExecutor` class in their own code, override the method creating the visitor so that it is wrapped in the decorator, and then use the new executor only when running tests. In addition to adding code, this introduces the undesirable case where the test code significantly differs from production code. This is particularly unattractive for the CARM team, which already uses a custom `InternalTransformationExecutor` which would need to be modified or replaced depending on whether the code is currently being executed inside of a test.

`TransformationCoverageModel`, and `NodeCoverageModel`. Unlike the Data classes, which intentionally store a minimal amount of data, the Model classes contain functions to retrieve both visited and unvisited nodes and to retrieve Eclipse resources representing modules or projects. The Model objects are then used to populate the QVTo Coverage View.

Although writing to and reading from disk may sound inefficient, it does not introduce significant overhead. In fact developers reported they noticed no additional overhead when using they tool. To assess this more quantitatively, we present a timing analysis as part of our evaluation in Section 12.2. Nonetheless, investigating additional communication mechanisms is left as future work.

### 11.4.2 Efficiently handling expression coverage

Within the `TransformationCoverageModel` objects, among other information stored are the lists of visited and unvisited expressions. Visited expressions are obtained from the Data objects. Unvisited expressions are obtained by traversing all children of the abstract syntax elements representing the QVTo module. If the expression does not exist in the list of visited expressions, then it is placed in the list of unvisited expressions. This presents two problems. First, not all expression types are visited by the QVTo visitor. For example, the left-hand side of assignment expressions (e.g. setting the value of a variable) is never visited because the QVTo interpreter simply overwrites the value previously stored in the variable, making it unnecessary to formally visit the variable beforehand. If not addressed, the coverage data is contradictory, both in the percentage calculation as well as the visual, because the left-hand side of an assignment would be red and the right hand side will be green, and the unvisited expression count would increase. Therefore it is necessary to explicitly ignore some expression types. Therefore, for this case, it checks if the parent expression is an assignment expression and whether the left child of the parent is equal to the expression currently being visited. Then it ignores the expression. In the current implementation, this is the only expression type ignored, though it is possible there are additional inconsistencies we have not yet encountered.

The second problem with constructing the lists of visited and unvisited expressions is that, as explained in Section 10.3, expressions are nested. Therefore, the ranges to highlight green and the ranges to highlight red may overlap, namely when the parent expression is visited but a child of that expression is not (e.g. a subclause of an `if` statement). To display an intuitive highlighting of code coverage to the developer, therefore, red highlighting must always take precedence over green highlighting. One possible solution to this is simply to render the green markers for visited expression nodes first and afterwards overlay the unvisited nodes with red markers. However, because of how Eclipse internally handles text markers, text markers are always placed asynchronously inside the editor. So, it is not possible to force red markers to appear after the green markers. The result is then that which marker is placed on top is random, and may even change every time the file is opened from the Coverage View.

Therefore, before displaying the markers, a new set of visited ranges is calculated by logically subtracting the unvisited ranges from the visited ranges. This is performed efficiently using a *range set* data structure implemented as an augmented binary tree. This data structure is designed to efficiently support the storage of arbitrary ranges. When a range is added which overlaps with a range already in the set, they are merged.

Ranges can also be subtracted, where subtracting a range contained within another range will split the parent range into two smaller ranges. A range set implementation is provided by Google's Guava library [103]. In our tool, we create an initial range set of visited nodes (where the range endpoints are integers representing the start and end characters of the expression node), build a range set of unvisited nodes, and then subtract each unvisited range from the visited range set. Afterwards, we can iterate through each set—now guaranteed to be exclusive—and add the red and green markers.

### 11.4.3 Accommodating CARM URIs

Cross-project builds, i.e. transformations which use modules located in other Eclipse projects, is not yet properly supported by the Eclipse QVTo engine (see bug report at [104]). Cross-project builds are however important since keeping all transformations within one project quickly leads to an unmaintainable code base. Therefore, the CARM team has added special support in their code to make cross-project builds possible by registering a new type of Eclipse uniform resource identifier (URI) which can be used to address other CARM projects. Because these URIs are not understood by the QVTo engine itself, however, the code coverage tool also could not read them. Therefore, a modification was made to the code coverage tool so that it too utilized the special addressing scheme. Specifically, a dependency was added between the coverage plugin and the plugin created by the CARM team containing the URI parser. Fixing cross-project builds and in turn removing this dependency is left as future work. Furthermore, because this is only necessary for the CARM URIs, this dependency is not included in the open source version of the tool.

### 11.4.4 Logging usage data

To support usage data collection, logging functionality was added to the coverage tool. To do so, an additional plugin (shown previously in Figure 11.2) was also developed. With this plugin, messages can be logged both to the console as well as to files on the user's local filesystem. Using the logging plugin, multiple usage events were collected by the coverage tool: a message signaling that a test run has been started, a message signaling an individual test case has completed, and a message signaling that a user has clicked on a transformation from the coverage view in order to browse the coverage highlighting for a specific file. For each message the date and time is also recorded. Usage data can then be retrieved simply by requesting the files from the tool users.

Since the types of data being collected are minimal, and because collecting the data from the developers is a manual step, we do not expect any impact on developer behavior. This was confirmed in a conversation with all of the CARM developers.

### 11.4.5 Changing coverage thresholds

Although the low and high coverage thresholds default to 30% and 90% respectively, support was also built in for tool users to change the coverage thresholds themselves. This support leverages the Eclipse preferences extension point, so the values can be set directly from the Eclipse preferences window. A screenshot of the new preferences pane

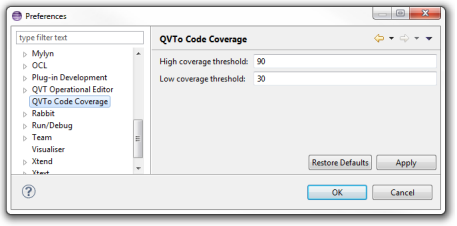


FIGURE 11.5: Coverage thresholds preferences pane in Eclipse

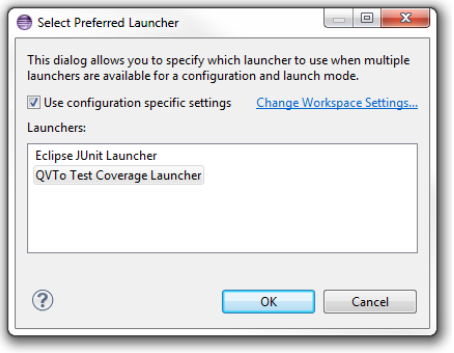


FIGURE 11.6: Selecting the QVTo coverage launcher

is displayed in Figure 11.5. The values are therefore settable per Eclipse environment according to the developer's preference. This is particularly important because developers may use the tool in different ways, and this allows users to customize the tool to maximize its usefulness for them personally. Moreover, it helps combat the potentially harmful expectation that coverage should always be 100%, as recommended by [99] in Section 10.3. The tool preferences screen also allows easy extension for additional user-settable preferences in the future.

### 11.4.6 Installing and launching the tool

As described in Section 11.2.1, the code coverage tool is implemented by extending the JUnit launch configuration of Eclipse. After tool installation, performed simply by adding the tool's update site to the Eclipse update manager, users can use the tool together with any JUnit launch. When the user runs a JUnit launch, they are automatically prompted whether they would like to use the JUnit launcher or the QVTo coverage launcher, as shown in Figure 11.6. Their choice can either be saved as a configuration-specific setting (convenient if there are other JUnit configurations where the user explicitly does not want coverage run) or globally in Eclipse (in which case every subsequent JUnit launch would use the QVTo launcher). Therefore both developers which run JUnit tests exclusively for QVTo, such as the case with our CARM team, as well as developers which maintain a number of different types of JUnit tests can be accommodated by the tool. If the user accidentally selects the QVTo launcher for a JUnit test that is not related to QVTo, the test runs normally. The tool therefore provides an almost effortless installation and requiring zero instructions to start using.

### 11.4.7 Open-sourcing

To maximize the usefulness of the tool, it has been made open source so that anyone can download the tool for free. Furthermore, since the source code is available, other developers can contribute new features to the tool themselves. This way, even though the tool here was developed with the CARM teams context in mind, other teams can customize the tool for their development procedures as well.

The tool code is available for browsing and download on Github [105]. Code is open sourced together with a *license*. For the coverage code, we have chosen to use the Eclipse Public License because it allows code to be freely used by others, but also allows linking to commercial libraries [106], which is often important for commercial entities hoping to use open source software in their own code. Moreover, it is the license used by the QVTo Eclipse implementation, so using this license can allow the coverage tool to be smoothly incorporated into the Eclipse project in the future. We hope, and per discussions with the Eclipse QVTo maintainers, that our tool becomes incorporated into the Eclipse project by the Mars release in 2015. More information on open source licenses is available at [107].

# Chapter 12

# Tool Evaluation and Future Work

In this chapter we describe the evaluation of the code coverage tool and directions for future work. As described in 9.3, we perform three methods of evaluation. The qualitative evaluation using group interviews together with the usage data analysis is presented in Section 12.1. The quantitative timing analysis is presented in Section 12.2. Finally, directions for future work for the tool developed here is presented in Section 12.3.

## 12.1 Qualitative Evaluation

Four CARM developers participated in the group evaluation interview since the remaining two were unavailable. Only one of the two absent had used the tool, who was followed-up with via email. The interview lasted approximately 45 minutes. Again a semi-structured interview was used, together with the interview guide from Figure 12.1.

The usage period lasted a total of seven weeks. During that period, the coverage tool was used on every QVTo test run by the CARM team. Specifically, usage data reported a total of 98 test runs (either a test set or test suite) were performed with the coverage during which 16,714 unit tests were executed.

The most common use case was to check which code in certain transformations was not touched, primarily using the highlighting overlay feature. Developers viewed the highlighting overlay according to the usage statistics 35 times. Notably, the tool was also used by a developer in the preparation of a *user story*. A user story is used in agile development teams to describe how to complete a specific feature or task of an agile sprint [108]. In the CARM group user stories are compiled as formal documents with specific steps needed to be taken to complete the feature. In preparing the user story, the developer ran the entire test suite and looked at the coverage for the modules he knew would be affected by the task. He then noted exactly which places in the modules were not covered by the test suite by inspecting the coverage overlay. Then, as part of the list of steps for the feature, he added an additional step stating that before implementation of the feature can begin, additional tests must be written to cover the parts of the transformation which were untested but would be affected by the feature. The developers also mentioned (without being prompted) that while browsing highlighting they could easily see which code may be dead code. Thus the second attribute intended to be addressed by the coverage tool is also at least partially fulfilled. However, the developers

- What are you general thoughts about the tool?
- What do you consider the most useful features?
- Of the coverage criteria displayed, which was most important or useful to you?
- What use cases did you use the tool for?
- How appropriate were the default coverage thresholds (30% and 90%)?
- Was there a performance impact?
- Will you continue using the tool and would you recommend it to others?
- Can you think of any cases where you would disable coverage while running unit tests?
- Do you feel like it improved quality? If yes, how? If not, what could be improved?
- Did you encounter any bugs?
- What additional features would you like to see in the tool?

FIGURE 12.1: Interview guide used for tool evaluation interview

also noted that it is difficult to tell if code is dead because of the transformation structure or if it is dead because of the metamodel structure (i.e. parts of the metamodel which are no longer used). Therefore, we identify adding metamodel coverage functionality to our tool as an area for future work, for instance leveraging the TraceVis approach [96] discussed in Section 9.1.

The developers agreed that the highlighting overlay was the most important feature of the tool, since there one can actually see which parts of code are not tested. The specific numbers displayed in the coverage view were not used often during the usage period, though the developers still asserted that quickly identifying which test cases have the lowest coverage is a useful and valid use case in the future. The developers identified another use case as using the coverage tool to report their confidence in the robustness of certain projects together with project documentation. The developers already stated however that they consider the tool "very useful", and that they will "absolutely continue using it". They also reported no noticeable different in test execution time. Installing the coverage tool has also been added to their team's "Way of working" document, already making it an official part of their development process.

The most useful coverage criteria according to the developers was the expression coverage, and they did not find the interpretation of the expression coverage statistic unintuitive (which was identified as a potential problem in Section 10.3.2). Next most useful was the Total function coverage. Although mapping, helper, and constructor coverage were considered less useful, the developers still asserted that it was nice to have the extra information there and at the very least was no hindrance to tool usage. One of the developers did note that additional types of coverage used by GPL coverage tools like proper statement coverage and branch coverage, would be nice to implement in the coverage tool. This developer however had not run any unit tests during the usage period

and said he suggested it because "if other tools have it there must be some benefit". So, considering that the developers were not utilizing the coverage statistics heavily yet, we consider this a lower priority feature.

The developers agreed that at least for their purposes, the high default threshold (above which the cover view cells are colored green) could be lowered to 80% since they consider anything above 80% to be well-covered. The low coverage could also be raised, since according to one the developers, "I am pretty sure we do not have that good of a test suite but there's almost no red", so a higher threshold feels more accurate. Their collective recommendation was to raise the lower threshold to 50%. They also expressed that rather than leaving the remaining cells white, yellow might be better, since intuitively that lies in between red and green. The developers also expressed that it would be useful if coverage thresholds could actually be specified on a per module basis, since certain CARM transformations are considered to be more critical than others, and should be held to higher coverage standards. They noted for instance that libraries ought to me more well-tested than transformations since many modules depend on them.

Finally, a number of additional features were identified by the developers as well. First, it would be nice of the tool could be run standalone via the command line, since one of their goals is to automate their test runs. Another feature would be adding export functionality which exported the coverage statistics displayed in the coverage view into for instance CSV, HTML, or XML formats.

### 12.1.1 Success

Given the positive feedback of the developers, we consider the tool evaluation very positive. When asked whether they felt the tool increased transformation quality, they replied, "of course. Now that we can actually measure the quality of the test suite, we know where we need to work to improve it". Combined with the feedback above, we therefore consider the tool to successfully increase quality in the context of the CARM team.

However, the tool should still be used by more teams and on more projects. Another threat to validity of this conclusion is, like in the survey used to evaluate the quality model in Chapter 7, that developers may speak more positively about the tool in order to not offend the interviewer. To combat this, it is recommended in future work that evaluation interviews be conducted by an independent party.

## 12.2 Quantitative Evaluation

Since the tool uses both interpreter instrumentation and disk access, there is inevitably some negative impact on performance. If this impact is large, it could cause developers to opt not to use the tool. Therefore, we perform a small quantitative evaluation of the tool with respect to execution performance. Specifically, we compared the test execution times for two real, production test sets maintained by the CARM team when using the QVTo coverage launcher versus the original JUnit launcher. The first test set ("Test Set A") was comprised of 28 individual tests, touching 7 QVTo modules containing (according to the QVTo coverage tool) a total of 3,959 expressions of which 2,767 were touched.

**Test Set A**

| Test # | Coverage (sec) No | Yes | % Change |
|---|---|---|---|
| 1 | 5.636 | 6.12 | 8.59 |
| 2 | 0.665 | 0.674 | 1.35 |
| 3 | 0.57 | 0.676 | 18.6 |
| 4 | 0.845 | 1.029 | 21.78 |
| 5 | 0.582 | 0.637 | 9.45 |
| 6 | 0.57 | 0.699 | 22.63 |
| 7 | 0.391 | 0.542 | 38.62 |
| 8 | 0.251 | 0.309 | 23.11 |
| 9 | 0.268 | 0.354 | 32.09 |
| 10 | 0.223 | 0.291 | 30.49 |
| 11 | 0.249 | 0.29 | 16.47 |
| 12 | 0.193 | 0.192 | -0.52 |
| 13 | 0.133 | 0.214 | 60.9 |
| 14 | 0.196 | 0.243 | 23.98 |
| 15 | 0.283 | 0.356 | 25.8 |
| 16 | 0.113 | 0.184 | 62.83 |
| 17 | 0.118 | 0.217 | 83.9 |
| 18 | 0.221 | 0.237 | 7.24 |
| 19 | 0.099 | 0.17 | 71.72 |
| 20 | 0.096 | 0.137 | 42.71 |
| 21 | 0.212 | 0.217 | 2.36 |
| 22 | 0.192 | 0.216 | 12.5 |
| 23 | 0.221 | 0.273 | 23.53 |
| 24 | 0.257 | 0.257 | 0 |
| 25 | 0.171 | 0.219 | 28.07 |
| 26 | 0.18 | 0.215 | 19.44 |
| 27 | 0.141 | 0.181 | 28.37 |
| 28 | 0.063 | 0.085 | 34.92 |
| Total | 13.143 | 15.265 | 16.15 |

**Test Set B**

| Test # | Coverage (sec) No | Yes | % Change |
|---|---|---|---|
| 1 | 5.36 | 5.738 | 7.05 |
| 2 | 0.693 | 0.783 | 12.99 |
| 3 | 0.558 | 0.739 | 32.44 |
| 4 | 0.558 | 0.795 | 42.47 |
| 5 | 0.415 | 0.371 | -10.6 |
| 6 | 0.544 | 0.532 | -2.21 |
| 7 | 0.45 | 0.641 | 42.44 |
| 8 | 0.404 | 0.379 | -6.19 |
| 9 | 0.413 | 0.377 | -8.72 |
| 10 | 0.14 | 0.312 | 122.86 |
| 11 | 0.237 | 0.315 | 32.91 |
| 12 | 0.249 | 0.321 | 28.92 |
| 13 | 0.279 | 0.298 | 6.81 |
| 14 | 0.21 | 0.237 | 12.86 |
| 15 | 0.221 | 0.293 | 32.58 |
| 16 | 0.254 | 0.255 | 0.39 |
| 17 | 0.214 | 0.252 | 17.76 |
| 18 | 0.234 | 0.29 | 23.93 |
| 19 | 0.174 | 0.207 | 18.97 |
| 20 | 0.27 | 0.298 | 10.37 |
| 21 | 0.533 | 0.67 | 25.7 |
| 22 | 0.245 | 0.354 | 44.49 |
| 23 | 0.404 | 0.562 | 39.11 |
| 24 | 0.243 | 0.294 | 20.99 |
| 25 | 0.182 | 0.351 | 92.86 |
| 26 | 0.2 | 0.315 | 57.5 |
| 27 | 0.075 | 0.136 | 81.33 |
| 28 | 0.228 | 0.261 | 14.47 |
| 29 | 0.212 | 0.393 | 85.38 |
| 30 | 0.164 | 0.252 | 53.66 |
| 31 | 0.189 | 0.219 | 15.87 |
| 32 | 0.021 | 0.028 | 33.33 |
| 33 | 0.231 | 0.307 | 32.9 |
| 34 | 0.257 | 0.407 | 58.37 |
| 35 | 0.383 | 0.547 | 42.82 |
| 36 | 0.198 | 0.22 | 11.11 |
| 37 | 0.19 | 0.258 | 35.79 |
| 38 | 0.241 | 0.281 | 16.6 |
| 39 | 0.179 | 0.215 | 20.11 |
| 40 | 0.268 | 0.305 | 13.81 |
| 41 | 0.234 | 0.25 | 6.84 |
| 42 | 0.186 | 0.213 | 14.52 |
| 43 | 0.187 | 0.251 | 34.22 |
| 44 | 0.631 | 0.942 | 49.29 |
| 45 | 0.228 | 0.303 | 32.89 |
| Total | 18.261 | 21.812 | 19.45 |

TABLE 12.1: Results from timing analysis performed on two production test sets

The second test set ("Test Set B") contained 45 tests with 4 QVTo modules and 3,023 total expressions of which 1,464 were touched. The timing results are extracted from the JUnit view inside Eclipse, which displays the time required in seconds per test up to three decimal places. All test runs were performed in the same environment.

The results of the timing analysis are presented in Table 12.1. Note that Test 1, 2, and so on are different, unique tests in the suite, and therefore expected to have different execution times. Although only one run is displayed for each set, times were checked to

make sure they were representative. In each run, the first test incurs significantly more time because the CARM test set performs initialization, for example registering required cross-project modules as described in Section 11.4.3. In general, we see that test runs take approximately 18% more time when using the coverage tool. Looking at individual test cases, we see that some test runs incur considerably more overhead than others. Inspecting the tests manually suggests that this variety of overhead between test cases is caused by a combination of number expressions/modules traversed (which depends on the input model), as well as the overhead caused by the coverage tool's file access, which in particular for very small input models is a large proportion of the original execution time. Some overhead even appeared as negative, which we explain by very slight variations in background processes on the system during execution.

By comparison, the EclEmma tool [100] commonly used for measuring Java code comparisons is expected to induce approximately 10% overhead [109], though this number, like ours, is highly test-case dependent, ranging from 5% to 30% in normal projects [110].

Furthermore, since the overhead incurred with using coverage amounts to only a couple seconds for a test set, we do not consider this to have an impact on tool usability. Even when considering the entire test suite, which requires approximately one and a half minutes to complete, the overhead is not enough to affect end-user behavior. The lack of impact on usability was further confirmed by the developers during the qualitative evaluation described in Section 12.1.

## 12.3 Future Work

There are many opportunities for future work with the code coverage tool. First, the suggestions from the developer evaluation should be addressed. Second, the communication mechanism between the Eclipse and Test VMs could potentially be improved. Cross-project builds should also be fixed in the QVTo core so that CARM libraries do not require special treatment. Additional patches could also be submitted for QVTo to make the inclusion of decorators even more transparent, since at the moment, it is still required to use the Java reflection API in order to add a third party decorator to the `VISITOR_DECORATORS` option.

Finally, in addition to the patches accepted into the QVTo core during this research and open sourcing the coverage tool, it is the hope of the first author and of the QVTo core maintainers that the coverage tool developed here also be incorporated into the QVTo core. Since that is a major commit and could not be included in the Luna release, the maintainers expressed that they hope this to be included in the Mars release planned for June 2015. Moreover, there is in general much work to be done in creating a full-fledged testing framework inside the QVTo core. This framework should provide a standardized way of writing and executing QVTo tests, which currently lacks from any of the QVTo implementations.

# Chapter 13

# Conclusions

To conclude this thesis, we review the contributions made by each of the two parts. In Part I, we addressed the question of how to assess QVTo model transformation quality. Using a synthesis approach, we constructed a quality model for QVTo transformations. This QVTo quality is model is the first major contribution of this research. Our synthesis approach and model evaluation relied heavily on empirical methods and developer participation in order to guarantee that the contents of the quality model were relevant to QVTo practitioners. Since this differs from how quality models have been developed in related work, we consider our approach and evaluation technique another contribution of this research. A set of best practices and difficulties gathered from expert interviews was also presented, which can be used by practitioners as a starting point for their own developer guidelines. These contributions were formulated inside a paper [19], which was accepted to QUATIC '14 [20] and named one of best in track.

In Part II the quality model attributes were scrutinized to identify methods to help developers create higher-quality transformations, addressing our second research question. A tool was selected for implementation, namely a QVTo test coverage tool. This tool, already integrated into the developer workflow at ASML, is the second major contribution of this research. The tool was evaluated by developers to have a positive impact on QVTo transformation quality, providing not only validation of the tool, but also of the QVTo quality model attributes which it addressed. This tool has also been made open-source so that QVTo practitioners can freely download and modify the tool. Other contributions of this part was a set of patches to the QVTo core that should make future QVTo tooling easier to develop and many directions for future work. Therefore contributions have been made in this thesis to academia, to ASML, and to the QVTo community. A paper describing the coverage tool contributions is planned to be submitted either to an industry track of ICSME [111] or to the Quality in MDE special issue of the Software Quality Journal by Springer [112].

# Bibliography


[1] Jos Warmer and Anneke Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] Victor R Basili. Software modeling and measurement: the Goal/Question/Metric paradigm. *Technical Report, University of Maryland*, 1992.

[3] Parastoo Mohagheghi and Vegard Dehlen. Developing a quality framework for model-driven engineering. In *Models in Software Engineering*, pages 275–286. Springer, 2008.

[4] Yuehua Lin, Jing Zhang, and Jeff Gray. A testing framework for model transformations. In *Model-driven software development*, pages 219–236. Springer, 2005.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[6] Marcel F van Amstel, Mark GJ van den Brand, and Phu H Nguyen. Metrics for model transformations. In *Proceedings of the Ninth Belgian-Netherlands Software Evolution Workshop (BENEVOL 2010), Lille, France (December 2010)*, 2010.

[7] OMG. MOF 2.0 Query/View/Transformation Spec. V1.1, 2011.

[8] Object management group (OMG), 2014. URL http://www.omg.org/.

[9] Radomil Dvorak. Model transformation with operational QVT, 2008. URL http://help.eclipse.org/juno/topic/org.eclipse.m2m.qvt.oml.doc/references/M2M-QVTO.pdf/.

[10] Pavle Guduric, Arno Puder, and Rainer Todtenhofer. A comparison between Relational and Operational QVT Mappings. In *Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference on*, pages 266–271. IEEE, 2009.

[11] Andrea Ciancone, Antonio Filieri, and Raffaela Mirandola. Mantra: Towards model transformation testing. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 97–105. IEEE, 2010.

[12] Andreas Rentschler, Qais Noorshams, Lucia Happe, and Ralf Reussner. Interactive visual analytics for efficient maintenance of model transformations. In *Theory and Practice of Model Transformations*, pages 141–157. Springer, 2013.

[13] Ramon Schiffelers, Wilbert Alberts, and Jeroen Voeten. Model-based specification, analysis and synthesis of servo controllers for lithoscanners. *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, 2012.

[14] ASML N.V., 2014. URL http://www.asml.com.

[15] Thomas (Tom) Stahl and Markus Voelter. *Model-driven software development*. John Wiley & Sons Chichester, 2006.

[16] Eugene Syriani and Jeff Gray. Challenges for addressing quality factors in model transformation. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 929–937. IEEE, 2012.

[17] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008.

[18] Barney G Glaser and Anselm L Strauss. *The discovery of grounded theory: Strategies for qualitative research*. Sociology Press, 1999.

[19] Christine Gerpheide, Ramon Schiffelers, and Alexander Serebrenik. A bottom-up quality model for QVTo. In *Quality of Information and Communications Technology (QUATIC), 2014 Ninth International Conference on the*. IEEE, 2014 (accepted).

[20] 9th international conference on the quality of information and communications technology, May 2014. URL http://2014.quatic.org.

[21] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[22] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45, 2003.

[23] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006.

[24] OMG. Object Constraint Language, 2012.

[25] Ivan Kurtev. State of the art of QVT: A model transformation language standard. In *Applications of Graph Transformations with Industrial Relevance*, pages 377–393. Springer, 2008.

[26] Sebastian Lehrig. *Assessing the quality of model-to-model transformations based on scenarios*. PhD thesis, MSc Thesis, University of Paderborn, Zukunftsmeile 1, 2012.

[27] Shekoufeh Kolahdouz-Rahimi, Kevin Lano, S. Pillay, J. Troya, and Pieter Van Gorp. Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming*, 85:5–40, 2014.

[28] Model transformation language, January 2014. URL http://en.wikipedia.org/wiki/Model_transformation_language.

[29] Siegfried Nolte. *QVT-operational mappings: Modellierung mit der Query views Transformation*. Springer, 2010.

[30] Pieter J Barendrecht. Modeling transformations using QVT Operational Mappings. Master's thesis, Technische Universiteit Eindhoven, 2010.

[31] Eclipse QVT Operational, January 2014. URL http://projects.eclipse.org/projects/modeling.mmt.qvt-oml.

[32] SmartQVT 0.2.2, November 2013. URL http://smartqvt.soft112.com/.

[33] Borland Together, November 2013. URL https://www.borland.com/products/together/.

[34] Eclipse modeling framework, January 2014. URL http://www.eclipse.org/modeling/.

[35] TIOBE coding standard framework TICS, January 2014. URL http://www.tiobe.com/index.php/content/products/tics/TICS_framework.html.

[36] Interview with Walfried Veldman, December 2013.

[37] Rudolf Ferenc, Péter Hegedűs, and Tibor Gyimóthy. Software product quality models. In Tom Mens, Alexander Serebrenik, and Anthony Cleve, editors, *Evolving Software Systems*, pages 65–100. Springer Berlin Heidelberg, 2014.

[38] ISO/IEC 9126-1:2001, 2014. URL http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749.

[39] Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: the elusive target [special issues section]. *Software, IEEE*, 13(1):12–21, 1996.

[40] Marcel F van Amstel. The right tool for the right job: assessing model transformation quality. In *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*, pages 69–74. IEEE, 2010.

[41] ISO/IEC 25010:2011, 2014. URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733.

[42] Marcel F van Amstel. *Assessing and Improving the Quality of Model Transformations*. PhD thesis, Technische Universiteit Eindhoven, 2012.

[43] Abraham H Maslow. *The Psychology of Science*. December 1966.

[44] Tracy Hall and Norman Fenton. Implementing effective software metrics programs. *Software, IEEE*, 14(2):55–65, 1997.

[45] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377. IEEE Computer Society, 2009.

[46] Daniel L Moody. Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. *Data & Knowledge Engineering*, 55(3):243–276, 2005.

[47] Hüseyin Ergin. Increasing the quality of model transformation with the use of design patterns. 2013.

[48] R Geoff Dromey. Cornering the chimera [software quality]. *Software, IEEE*, 13(1): 33–43, 1996.

[49] Jagdish Bansiya and Carl G Davis. A hierarchical model for object-oriented design quality assessment. *Software Engineering, IEEE Transactions on*, 28(1):4–17, 2002.

[50] Lukman Ab Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Software & Systems Modeling*, pages 1–26, 2013.

[51] Siw Elisabeth Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 10–pp. IEEE, 2005.

[52] Carolyn B Seaman. Qualitative methods in empirical studies of software engineering. *Software Engineering, IEEE Transactions on*, 25(4):557–572, 1999.

[53] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33:2004, 2004.

[54] Google scholar, December 2013. URL http://scholar.google.com/intl/en/scholar/about.html.

[55] QVT tutorial, November 2013. URL http://www.redpanda.nl/index.php?p=tutorial.

[56] Markus Voelter. Best practices for DSLs and model-driven development. *Journal of Object Technology*, 8(6):79–102, 2009.

[57] Aditya Agrawal, Attila Vizhanyo, Zsolt Kalmar, Feng Shi, Anantha Narayanan, and Gabor Karsai. Reusable idioms and patterns in graph transformation languages. *Electronic Notes in Theoretical Computer Science*, 127(1):181–192, 2005.

[58] Maria-Eugenia Iacob, Maarten WA Steen, and Lex Heerink. Reusable model transformation patterns. In *Enterprise Distributed Object Computing Conference Workshops, 2008 12th*, pages 1–10. IEEE, 2008.

[59] Paul Gniesser. Refactoring support for ATL-based model transformations. Master's thesis, Faculty of Informatics-Vienna University of Technology, 2012.

[60] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[61] Louis M Rose, Markus Herrmannsdoerfer, Steffen Mazanek, Pieter Van Gorp, Sebastian Buchwald, Tassilo Horn, Elina Kalnina, Andreas Koch, Kevin Lano, Bernhard Schätz, and Manuel Wimmer. Graph and model transformation tools for model migration. *Software & Systems Modeling*, pages 1–37, 2012.

[62] Transformation tool contest, December 2013. URL http://www.transformation-tool-contest.eu.

[63] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Werner Retschitzegger, Wieland Schwinger, and Gerti Kappel. Reality check for model transformation reuse: The ATL transformation zoo case study. *2nd Workshop on the Analysis of Model Transformations (AMT) @ MODELS'13*, 1077, 2013.

[64] ATL transformations zoo, May 2014. URL http://www.eclipse.org/atl/atlTransformations/.

[65] Marcel F van Amstel, Steven Bosems, Ivan Kurtev, and Luís Ferreira Pires. Performance in model transformations: experiments with ATL and QVT. In *Theory and Practice of Model Transformations*, pages 198–212. Springer, 2011.

[66] QVT project organization, January 2014. URL http://www.eclipse.org/forums/index.php/t/487579/.

[67] QVT Operational Mappings, January 2014. URL http://adolfosbh.blogspot.nl/2013/10/qvt-operational-mappings.html.

[68] Lucia Kapová, Thomas Goldschmidt, Steffen Becker, and Jörg Henss. Evaluating maintainability with code metrics for model-to-model transformations. In *Research into Practice–Reality and Gaps*, pages 151–166. Springer, 2010.

[69] Phu hong Nguyen. Quality analysis of model transformations. Master's thesis, Technische Universiteit Eindhoven, 2010.

[70] Elena Planas, Jordi Cabot, and Cristina Gómez. Two basic correctness properties for ATL transformations: Executability and coverage. In *3rd International Workshop on Model Transformation with ATL, Zurich, Switzerland*, 2011.

[71] Gehan MK Selim, James R Cordy, and Juergen Dingel. Model transformation testing: The state of the art. In *Proceedings of the First Workshop on the Analysis of Model Transformations*, pages 21–26. ACM, 2012.

[72] Jacqueline A McQuillan and James F Power. White-box coverage criteria for model transformations. *Model Transformation with ATL*, page 63, 2009.

[73] Marcel F van Amstel and Mark GJ van den Brand. Model transformation analysis: Staying ahead of the maintenance nightmare. In *Theory and Practice of Model Transformations*, pages 108–122. Springer, 2011.

[74] Andreas Rentschler, Qais Noorshams, Lucia Happe, and Ralf Reussner. Interactive visual analytics for efficient maintenance of model transformations. In *ICMT*, volume 7909 of *LNCS*, pages 141–157. Springer, 2013.

[75] Richard F Paige and Dániel Varró. Lessons learned from building model-driven development tools. *Software & Systems Modeling*, 11(4):527–539, 2012.

[76] Eclipse community forum QVT-OML, January 2014. URL http://www.eclipse.org/forums/index.php/f/244.

[77] Missile Defense Agency. Department of defense documentation of verification, validation and accreditation (VVA) for models and simulations. 2008.

[78] Naomi Oreskes. Evaluation (not validation) of quantitative models. *Environmental Health Perspectives*, 106(Suppl 6):1453, 1998.

[79] Graeme Shanks. Conceptual data modelling: an empirical study of expert and novice data modellers. *Australasian Journal of Information Systems*, 4(2), 2007.

[80] Daniel L Moody, Guttorm Sindre, Terje Brasethvik, and Arne Sølvberg. Evaluating the quality of process models: Empirical testing of a quality framework. In *Conceptual Modeling—ER 2002*, pages 380–396. Springer, 2003.

[81] Daniel L Moody. Measuring the quality of data models: an empirical evaluation of the use of quality metrics in practice. In *ECIS*, pages 1337–1352, 2003.

[82] Jon A Krosnick and Leandre R Fabrigar. Designing rating scales for effective measurement in surveys. *Survey measurement and process quality*, pages 141–164, 1997.

[83] William L Rankin and Joel W Grube. A comparison of ranking and rating procedures for value system measurement. *European Journal of Social Psychology*, 10 (3):233–246, 1980.

[84] Duane F Alwin and Jon A Krosnick. The measurement of values in surveys: A comparison of ratings and rankings. *Public Opinion Quarterly*, 49(4):535–552, 1985.

[85] Rob Johns. Likert items and scales. *Survey Question Bank: Methods Fact Sheet*, 1, 2010.

[86] J Jackson Barnette. Effects of stem and likert response option reversals on survey internal consistency: If you feel the need, there is a better alternative to using those negatively worded stems. *Educational and Psychological Measurement*, 60 (3):361–370, 2000.

[87] Clifford E Lunneborg. Book review: Psychometric theory: Second edition jum c. nunnally new york: Mcgraw-hill, 1978, 701 pages. *Applied Psychological Measurement*, 3(2):279–280, 1979.

[88] Ron Garland. The mid-point on a rating scale: Is it desirable. *Marketing Bulletin*, 2(1):66–70, 1991.

[89] Mailing list: qvto-dev, January 2014. URL https://dev.eclipse.org/mailman/listinfo/qvto-dev.

[90] Alan Agresti. *Categorical data analysis*, volume 359. John Wiley & Sons, 2002.

[91] Maurits Clemens Kaptein, Clifford Nass, and Panos Markopoulos. Powerful and consistent analysis of likert-type ratingscales. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2391–2394. ACM, 2010.

[92] Michael Haber and Huiman X Barnhart. Coefficients of agreement for fixed observers. *Statistical methods in medical research*, 15(3):255–271, 2006.

[93] Kendall's coefficient of concordance, 2014. URL http://pqstat.com/?mod_f=Kendall_W.

[94] Gaj Vidmar and Nino Rode. Visualising concordance. *Computational Statistics*, 22(4):499–509, 2007.

[95] Stanley A Mulaik. *The foundations of factor analysis*, volume 88. McGraw-Hill New York, 1972.

[96] Marcel F van Amstel, Mark GJ van den Brand, and Alexander Serebrenik. Traceability visualization in model transformations with TraceVis. In *Theory and Practice of Model Transformations*, pages 152–159. Springer, 2012.

[97] Barbara Kitchenham. Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods. *ACM SIGSOFT Software Engineering Notes*, 21(1):11–14, 1996.

[98] EMFCompare, January 2014. URL http://www.eclipse.org/emf/compare/.

[99] Brian Marick. How to misuse code coverage. In *Proceedings of the 16th Interational Conference on Testing Computer Software*, pages 16–18, 1999.

[100] EclEmma: Java code coverage for eclipse, January 2014. URL http://www.eclemma.org/.

[101] Github: Eclipse QVTo repository, May 2014. URL https://github.com/eclipse/qvto.

[102] Eclipse bugs by bugzilla, May 2014. URL https://bugs.eclipse.org/bugs/.

[103] Guava: Google core libraries for java 1.6+, May 2014. URL https://code.google.com/p/guava-libraries/.

[104] Bug 433937 — add support for libraries from other projects, May 2014. URL https://bugs.eclipse.org/bugs/show_bug.cgi?id=433937.

[105] Eclipse plugin for measuring QVTo test coverage, May 2014. URL https://github.com/phoxicle/qvto-coverage.

[106] Licenses, May 2014. URL http://choosealicense.com/licenses/eclipse/.

[107] Licenses, May 2014. URL http://choosealicense.com/licenses/.

[108] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. 2001.

[109] EclEmma: Control flow analysis for java methods, May 2014. URL http://www.eclemma.org/jacoco/trunk/doc/flow.html.

[110] Java code coverage: Reasons for huge performance impact, May 2014. URL http://comments.gmane.org/gmane.comp.java.jacoco.user/66.

[111] 30th international conference on software maintenance and evolution, May 2014. URL http://www.icsme.org.

[112] Software quality journal: Call for papers, May 2014. URL http://www.springer.com/cda/content/document/cda_downloaddocument/SQJO_CfP_QMDE.pdf.

# Appendix A

# Pre-interview Questionnaire

- What type of education have you had (level and subject)?
- What is your position at ASML and how long have you had it?
- Which other related positions have you had (at ASML or other companies)?
- What do you usually work on? For example, writing model transformations, which languages you work in, etc.
- How many people do you usually work with and what is your role in the team?
- How often do you work with QVTo specifically (# hours/week)?
- How strong do you consider yourself as a QVTo programmer?

# Appendix B

# Scripts for Validation Survey Processing

## B.1 reformat.py

This script produces individual files for each survey respondent's answers that can easily be read by a reviewer.

```
import csv
import re

# CONFIG
FILE_PATH = "C:\Users\cgerphei\Project\Docs\Surveys\QVTo_Public_Reacties_AB.csv"
HAS_EXPERIENCE_COL = True

#INTERNAL
OUTPUT_FILE_DIR = "C:\Users\cgerphei\Project\Docs\Surveys\Formatted\\"
OUTPUT_FILE_SUFFIX = "_formatted_reactie.html"

TIME_COL = 0
NAME_COL = 1
EXP_COL = 2

QUESTION_REGEX = re.compile('(.*)\[(.*)\]')



contents = ""

# Utils

def output_date(col):
        global contents
        contents += format_subheader(col)

def output_name(col):
        global contents
        contents += format_header(col)

def output_exp(col):
        global contents
        contents += format_p(col)
```

```
def output_question(col):
        global contents
        if not col: # Comments
                contents += format_plain("Comments:")
        match = QUESTION_REGEX.match(col)
        if match:
                question = match.group(1)
                quality = match.group(2)
                if quality == "Functionality":
                        contents += format_subheader(question)
                contents += format_plain(quality + ": ")
        else:
                contents += format_subheader(col)

def output_answer(col):
        global contents
        contents += format_plain(col)
        contents += format_break()

def format_header(str):
        return "<h1>" + str + "</h1>"

def format_subheader(str):
        return "<h4>" + str + "</h4>"

def format_p(str):
        return "<p>" + str + "</p>"

def format_plain(str):
        return str

def format_break():
        return "<br/>"

# Main


with open(FILE_PATH, 'rb') as csvfile:
        reader = csv.reader(csvfile, delimiter=',')
        header = ""
        for row in reader:
                # Build contents
                contents = ""
                current_name = ""
                if not header:
                        header = row
                else:
                        for i in range(len(row)):
                                col = row[i]
                                if i == TIME_COL:
                                        output_date(col)
                                elif i == NAME_COL:
                                        output_name(col)
                                        current_name = col
                                elif HAS_EXPERIENCE_COL and i == EXP_COL:
                                        output_question(header[i])
                                        output_exp(col)
                                else:
                                        output_question(header[i])
                                        output_answer(col)
                # Write file
```

```
                file_name = OUTPUT_FILE_DIR + current_name + OUTPUT_FILE_SUFFIX
                output_file = open(file_name, "w")
                output_file.write(contents)
                output_file.close()
```

## B.2 csvify.py

This script converts the Likert answers to a range and reformats the CSV to a format more suitable for import into R.

```
import csv, re

# CONFIG
FILE_PATH = "C:\Users\cgerphei\Project\Docs\Surveys\QVTo_Public_Reacties_AB.csv"
OUTPUT_FILE = "C:\Users\cgerphei\Project\Docs\Surveys\Processed\processed_combined.csv"
APPEND = True

#INTERNAL

TIME_COL = 0
NAME_COL = 1
EXP_COL = 2

TEXT2NUM = {
      "Strongly decreases" : -3,
      "Decreases" : -2,
      "Somewhat decreases" : -1,
      "Has little/no effect" : 0,
      "Somewhat increases" : 1,
      "Increases" : 2,
      "Strongly increases" : 3,
      "NULL" : "NULL"
}

QUESTION_REGEX = re.compile('\d+\. In your opinion, '
    + 'what effect does (.*) have on a transformation\? \[(.).*\]')
#QUESTION_REGEX = re.compile('(.*)')

header = []

# Format helpers

def format_question(col):
        match = QUESTION_REGEX.match(col)
        if match:
                metric = match.group(1)
                quality_letter = match.group(2)
                return re.sub(r'\W+', '', metric) + "_" + quality_letter

def format_name(col):
        return col

def format_answer(col):
        return TEXT2NUM[col]



# Row Helpers
```

```
def construct_header(row):
        cells = []
        for i in range(len(row)):
                col = row[i]
                if i == TIME_COL:
                        continue
                elif i == NAME_COL:
                        cells.append("name")
                elif i == EXP_COL:
                        continue
                elif not col or col=="That's it!": # comments
                        continue
                else:
                        cells.append(format_question(col))
        return cells

def construct_row(row):
        cells = []
        for i in range(len(row)):
                col = row[i]
                if i == TIME_COL:
                        continue
                elif i == NAME_COL:
                        cells.append(format_name(col))
                elif i == EXP_COL:
                        continue
                elif (not header[i]) or header[i]=="That's it!": # Comments column
                        continue
                else:
                        cells.append(format_answer(col))
        return cells

# Main

with open(FILE_PATH, 'rb') as infile:
        reader = csv.reader(infile, delimiter=',')
        mode = 'w'
        if APPEND:
                mode = 'a'
        with open(OUTPUT_FILE, mode) as outfile:
                writer = csv.writer(outfile, delimiter=',')
                for row in reader:
                        if not header:
                                header = row
                                if not APPEND:
                                        writer.writerow(construct_header(row))
                        else:
                                writer.writerow(construct_row(row))
```

## B.3 colorize.py

This script produces colorful output representing the survey responses that can be used to get an overall view of the survey responses.

```
import csv

# CONFIG
```

```
FILE_PATH = "C:\Users\cgerphei\Project\Docs\Surveys\QVTo_Public_Reacties_wMissing.csv"
INCLUDE_NAMES = False
ALT_CSS = False
OUTPUT_FILE = "C:\Users\cgerphei\Project\Docs\Surveys\colorized_public.html"

#INTERNAL

TIME_COL = 0
NAME_COL = 1
EXP_COL = 2

TEXT_NEG3 = "Strongly decreases"
TEXT_NEG2 = "Decreases"
TEXT_NEG1 = "Somewhat decreases"
TEXT_0 = "Has little/no effect"
TEXT_1 = "Somewhat increases"
TEXT_2 = "Increases"
TEXT_3 = "Strongly increases"
TEXT_EXEMPT = "NULL"

header = []
contents = ""

# Format helpers

def format_begin_table():
        return "<table>"

def format_end_table():
        return "</table>"

def format_begin_row():
        return "<tr>"

def format_end_row():
        return "</tr>"

def format_plain(str):
        return format_cell("",str)

def format_cell(style, str=" "):
        return "<td class='" + style + "'>" + str + "</td>"

def format_header():
        str = "<html><head><link rel='stylesheet' href='colorize.css' />"
        if ALT_CSS:
                str += "<link rel='stylesheet' href='colorize_alt.css' />"
        str += "</head><body>"
        return str

def format_end():
        return "</body>"

# Output Helpers

def output_begin():
        global contents
        contents += format_header()
        contents += format_begin_table()

def output_end():
        global contents
```

```
        contents += format_end_table()
        # Legend
        contents += format_begin_table()
        contents += format_begin_row() + format_cell("neg3", TEXT_NEG3) + format_end_row()
        contents += format_begin_row() + format_cell("neg2", TEXT_NEG2) + format_end_row()
        contents += format_begin_row() + format_cell("neg1", TEXT_NEG1) + format_end_row()
        contents += format_begin_row() + format_cell("zero", TEXT_0) + format_end_row()
        contents += format_begin_row() + format_cell("pos1", TEXT_1) + format_end_row()
        contents += format_begin_row() + format_cell("pos2", TEXT_2) + format_end_row()
        contents += format_begin_row() + format_cell("pos3", TEXT_3) + format_end_row()
        contents += format_begin_row() + format_cell("exempt", TEXT_EXEMPT) + format_end_row()
        contents += format_end_table()
        contents += format_end()

def output_header(row):
        global contents
        for i in range(len(row)):
                col = row[i]
                if not col: # comments
                        continue
                elif i == NAME_COL and not INCLUDE_NAMES:
                        continue
                elif i == EXP_COL and not INCLUDE_NAMES:
                        continue
                contents += format_plain(col)

def output_row(row):
        global contents
        contents += format_begin_row()
        for i in range(len(row)):
                col = row[i]
                if i == TIME_COL:
                        contents += format_plain(col)
                elif i == NAME_COL:
                        if INCLUDE_NAMES:
                                contents += format_plain(col)
                elif i == EXP_COL:
                        if INCLUDE_NAMES:
                                contents += format_plain(col)
                elif not header[i]: # Comments colum
                        continue
                else:
                        if col == TEXT_NEG3:
                                contents += format_cell("neg3")
                        elif col == TEXT_NEG2:
                                contents += format_cell("neg2")
                        elif col == TEXT_NEG1:
                                contents += format_cell("neg1")
                        elif col == TEXT_0:
                                contents += format_cell("zero")
                        elif col == TEXT_1:
                                contents += format_cell("pos1")
                        elif col == TEXT_2:
                                contents += format_cell("pos2")
                        elif col == TEXT_3:
                                contents += format_cell("pos3")
                        elif col == TEXT_EXEMPT:
                                contents += format_cell("exempt")
        contents += format_end_row()

# Main
```

```
with open(FILE_PATH, 'rb') as csvfile:
        reader = csv.reader(csvfile, delimiter=',')
        for row in reader:
                # Build contents
                if not header:
                        header = row
                        output_begin()
                        output_header(row)
                else:
                        output_row(row)
        output_end()

# Write file
output_file = open(OUTPUT_FILE, "w")
output_file.write(contents)
output_file.close()
```

## B.4 analyze.r

This script produces colorful output representing the survey responses that can be used to get an overall view of the survey responses.

```
###
# Config
###
path = "C:\\Users\\cgerphei\\Project\\Docs\\Surveys\\Processed\\processed_combined.csv"
boxplotfile = "C:/Users/cgerphei/Project/Docs/Surveys/Processed/boxplot.png"

###
# Prep
###

df = read.csv(path, header=TRUE)

# Set row names
rownames(df) <- df[,1]
df[,1] <- NULL

# Convert contents to numeric
df = data.frame( sapply(df, function(x) as.numeric(as.character(x))) )

# More intuitive representation has columns for each response
df = t(df)
df_raw = df # Just a copy

###
# Main
###

# Sensitivity! Drop each column once and see what happens
# names_before = total_names
#df = subset(df, select = -15)
# setdiff(names_before,total_names)


# Count of NAs
length(which(is.na(df)))
```

```
# Count numbers of 3s ad -3s total
table(df)
# Now per person
#apply(df_raw,2,table)

# Combine 2,3 and -2,-3
#df[which(df==3)] = 2
#df[which(df==-3)] = -2

# Box plot
#png(boxplotfile, width=5000, height=1000, res=120)
#par(mar=c(20,5,1,1))
#boxplot(df_raw, las=2)
#dev.off()


# Separate ASML (first 6 columns) from nonASML
df_asml = df[,1:6]
df_public = df[,7:15]


# Medians and ranges

get_ranges = function(df) {
    df = data.frame(
        # We ignore missing
        mymedian = apply(df,1, median, na.rm = TRUE),
        x25th = apply(df,1,quantile, probs=c(0.25), na.rm=TRUE),
        x75th = apply(df,1,quantile, probs=c(0.75), na.rm=TRUE),
        diffq = apply(df, 1, function(df) {
            quantile(df, probs=c(0.75), na.rm=TRUE) - quantile(df, probs=c(0.25), na.rm=TRUE)
        })
    )
}



df_ranges = get_ranges(df)
df_ranges_asml = get_ranges(df_asml)
df_ranges_public = get_ranges(df_public)


###
# Q1: How much do people agree with each other?
###

# How many questions have 0 or 1 range (max-min)?
#length(which(df_ranges$myrange==0))
#length(which(df_ranges$myrange==4)) # Disagreement

# Eliminate outliers
#length(which(df_ranges$diffq==0))
#length(which(df_ranges$diffq==4)) # Disagreement

# Graphically
#barplot(table(df_ranges$myrange))
barplot(table(round(df_ranges$diffq)))

# Kendall coeff of concordance W
# library(vegan)
# kendall.global(df)
```

```
# ICC
#library(psych)
# ICC1 is absolute agreement
#ICC(df, missing=FALSE)

# Check agreement if only diffq <= 1 included
df_1 = df[which(df_ranges$diffq <= 1),]
df_1_range = df[which(df_ranges$myrange <= 1),]
# Measure W

# Pairwise correlations
# mycors = cor(df, use='p', method="kendall")
# library(correlate)
# corrplot(mycors, method="circle", type="upper")



# Extract individual qualities

extract = function(df, num) {
    i = 1:nrow(df)
    df[i%%4==num,]
}

df_f = extract(df,1)
df_u = extract(df,2)
df_p = extract(df,3)
df_m = extract(df,0)

df_ranges_f = extract(df_ranges,1)
df_ranges_u = extract(df_ranges,2)
df_ranges_p = extract(df_ranges,3)
df_ranges_m = extract(df_ranges,0)


###
# Q2: Which attributes were most (least) agreed upon?
###

# Sort by range (which are best agreed upon) then medians (which are best)

sort_by_range = function(df) {
    df[order(df$myrange, rev(df$mymedian)),]
}

sort_by_diffq = function(df) {
    df[order(df$diffq, rev(df$mymedian)),]
}

#sort_by_range(df_ranges)

# Plot median versus diffq
#plot(jitter(df_ranges$mymedian), jitter(df_ranges$diffq))

###
# Q3: Were the attributes as a whole thought to increase quality?
###


# Histogram of answer totals
#barplot(table(df))
```

```
#barplot(table(df_u))


###
# Q4: Which attributes increase each quality goal most?
###

sort_by_median = function(df) {
    df[order(df$mymedian,decreasing=TRUE),]
}

# Most important attributes
#sort_by_median(df_ranges_public)

make_model = function(df) {
    df = df[which(df$mymedian >= 1),]
    df = df[which(df$x25th >= 0),]
}

model_total = make_model(df_ranges)
model_asml = make_model(df_ranges_asml)
model_public = make_model(df_ranges_public)

# Find differences between ASML/Public models
total_names = rownames(model_total)
asml_names = rownames(model_asml)
public_names = rownames(model_public)
# In public, not ASML
setdiff(public_names,asml_names)
# In ASML, not public
setdiff(asml_names,public_names)

# fail sensitivity
#setdiff(names_before,total_names)
# added with sensitivity
#setdiff(total_names,names_before)
```

# Appendix C

# Developing the Coverage Tool

This appendix describes how one can continue development of the coverage tool. Before reading this, it is best to read the chapter on tool implementation first (Chapter 11).

To develop the coverage tool, three Eclipse environments should be used:

- A development environment: In this environment, all plugins from the coverage tool are imported as well as the QVTo plugins imported from source. Here, the tool functionality can be altered. The tool is then tested by starting a new Eclipse instance from the `plugin.xml` file.
- A builder environment: In this environment, the plugins can be versioned and then built inside the update site. This environment should be identical to the environment where the plugins will eventually be installed by end users.
- A test environment: This environment is used for final testing of the update site and included plugins. The update site should be added to this installation and plugins should be installed and then tested as an end user. With this environment one can verify that the update site is installing all plugins properly.

In the development environment the plugin projects `de.phei.qvto.coverage`, `de.phei.qvto.coverage.common`, `de.phei.qvto.coverage.runtime`, and `de.phei.qvto.logging` can be imported. There changes can be made, and then tested by starting a runtime Eclipse instance from the `plugin.xml` of one of the plugins. If changes are also required in the QVTo engine, then that project can be imported from source using the "Plug-ins" tab inside the Plugin-in Development perspective. When launching the runtime instance, any QVTo projects imported into the workspace will automatically be used instead of the originals to allow for easy testing. If the QVTo engine changes must be used while the tests are running in the runtime instance, then the QVTo plugin should also be exported from the development environment and then set manually inside the build path of the Java project in the runtime instance.

Once the plugins are in the desired state, the builder environment can be opened. This environment should have the plugin projects listed above as well as the projects `de.phei.qvto.coverage.feature` and `de.phei.qvto.coverage.updatesite` imported in the workspace. The version numbers of any modified projects should be updated in the

`plugin.xml` of the project. Assuming at least one plugin was modified, then the version of the feature project should also be increased. The feature with the higher version should then replace the old one inside the update site project. The update site can then be built by opening the `site.xml` and running "Build all". If changes were also added in the QVTo engine, then that project as well as the `de.phei.qvto.patch` project. The version should be incremented in the patch project and (re)added to the update site before building the update site.

The update site can then be added as a local update site inside the test environment. After updating, QVTo JUnit tests can be run with the updated plugins as a normal user would run them.

## C.1 Developing similar tools

Much of the coverage tool infrastructure can be reused for other developer tools. Here we consider the case of a profiler tool. We assume that QVTo transformations are executed in a standalone context, therefore there exists a class which runs transformations with a given input and the desired result is to know how long executions took inside the transformation, for instance the execution times of each mapping.

Necessary setup can still be done with a shell class, like in the coverage tool. However, because the Java launch configuration is being used instead of the JUnit launch configuration, the Java launch delegate should be extended instead. The code required inside the launch configuration will be nearly identical.

A new decorator should be created which will record execution times rather than coverage data. This decorator can also be nearly identical to the `QVTOCoverageDecorator`, except that it will make use of new profiler Data objects instead of coverage Data objects and it should implement the `genericPostVisitAST()` method in addition to the `genericPreVisitAST()` method so that logic can be added to calculate elapsed time. Within those methods, different behavior can be implemented for different AST node types (e.g. mappings) if desired. The profiler Data object classes should also be implemented, mirroring the coverage Data objects. These objects should however should store timing information for each node rather than just touched versus not touched. Profiler Model objects should also be created similar to the coverage Model objects.

Finally, a new View should be created to display the profiler information to the user, similar to the Coverage View. This view should be shown when the transformation has finished executing. The internal logic used by JUnit's `TestRunListener` class can likely be mimicked to determine when to trigger the view.