Software Qual J (2016) 24:797–834
DOI 10.1007/s11219-015-9280-8

Assessing and improving quality of QVTo model transformations

Christine M. Gerpheide[1] · Ramon R. H. Schiffelers[2] · Alexander Serebrenik[1]

Published online: 3 June 2015
© The Author(s) 2015. This article is published with open access at Springerlink.com

Abstract We investigate quality improvement in QVT operational mappings (QVTo) model transformations, one of the languages defined in the OMG standard on model-to-model transformations. Two research questions are addressed. First, how can we assess quality of QVTo model transformations? Second, how can we develop higher-quality QVTo transformations? To address the first question, we utilize a bottom–up approach, starting with a broad exploratory study including QVTo expert interviews, a review of existing material, and introspection. We then formalize QVTo transformation quality into a QVTo quality model. The quality model is validated through a survey of a broader group of QVTo developers. We find that although many quality properties recognized as important for QVTo do have counterparts in general purpose languages, a number of them are specific to QVTo or model transformation languages. To address the second research question, we leverage the quality model to identify developer support tooling for QVTo. We then implemented and evaluated one of the tools, namely a code test coverage tool. In designing the tool, code coverage criteria for QVTo model transformations are also identified. The primary contributions of this paper are a QVTo quality model relevant to QVTo practitioners and an open-source code coverage tool already usable by QVTo transformation developers. Secondary contributions are a bottom–up approach to building a quality model, a validation approach leveraging developer perceptions to evaluate quality properties, code test coverage criteria for QVTo, and numerous directions for future research and tooling related to QVTo quality.

✉ Christine M. Gerpheide
christine.ger@phei.de

Ramon R. H. Schiffelers
r.r.h.schiffelers@tue.nl; ramon.schiffelers@asml.com

Alexander Serebrenik
a.serebrenik@tue.nl

[1] Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

[2] ASML N.V., De Run 6501, 5504 DR Veldhoven, The Netherlands

Keywords Software quality · QVTo · Model transformations · Quality model · Developer tooling · Test coverage

1 Introduction

Model-driven engineering (MDE) can be used to develop highly reliable software, offering benefits from analysis to code generation (Stahl and Voelter 2006). In MDE, models are created by domain experts and then transformed into other models or code using model transformations. One language for implementing model transformations is QVT operational mappings, aka. QVTo, specified in the 2007 Object Management Group (OMG) standard for model-to-model transformation (MMT) languages (OMG 2011).

As a standard, QVTo is used today in academia (e.g., onl 2014g) and industry (e.g., France Telecom 2014). In particular, ASML (onl 2014a), the leading provider of complex lithography systems for the semiconductor industry, uses QVTo as its primary language for implementing MMTs. Currently ASML has tens of thousands of lines of QVTo code supporting activities from real-time schedule analysis to servo controller initialization during machine start-up (Schiffelers et al. 2012).

While for general purpose languages (GPLs) developers have built up many common notions to judge whether a piece of code is of high or low quality, such agreed-upon best practices and quality indicators do not yet exist for QVTo. Given the many specific language features of QVTo (outlined in Sect. 2.1), it is even unclear whether quality properties 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 (Syriani and Gray 2012). Moreover, the lack of *tooling* embodying transformation quality is identified as one of the biggest limitations in developing high-quality model transformations in practice today (Syriani and Gray 2012).

Therefore, in this research we address two research questions. First, how can we assess quality of QVTo model transformations? Here, we investigate what code properties, best practices, and in general quality concerns currently exist for QVTo. To maximize the relevance of our research to industry, a focus on practitioners is maintained. We therefore explicitly adopt a *pragmatist* stance (Easterbrook et al. 2008), which states that knowledge is judged by how useful it is for solving practical problems. This stance drives many of our design decisions as well as our validation strategies. We use the data gathered to address our second research question: how can we develop higher-quality model transformations? Here, we employ our quality model to identify opportunities for developer support tooling. We then develop a code test coverage tool for QVTo and evaluate its success in helping developers improve quality of QVTo transformations.

We begin with an introduction to QVTo and software quality in Sect. 2. To guarantee that the notions of quality we identify are relevant for QVTo practitioners, we follow a *bottom–up approach*, described in Sect. 3, starting with a broad exploratory study including expert interviews, a review of existing material, and introspection. We then formalize the exploratory study results into a quality model for QVTo transformations, presented along with a selection of best practices and difficulties in Sect. 3.3. The validation of our quality model is presented in Sect. 3.4. We then describe potential directions for tooling based on our quality model in Sect. 4. The development approach and requirements of our chosen tool are described in Sect. 4.1 and its implementation is

elaborated in Sect. 4.3. An evaluation of the tool is presented is Sect. 4.4. Finally, we discuss additional related work in Sect. 5 and conclusions and future work in Sect. 6.

The primary contribution of this paper is a QVTo quality model relevant to QVTo practitioners and an open-source code coverage tool already able to be effectively used by QVTo transformation developers to develop higher-quality transformations. Secondary contributions are first our bottom–up approach to building a quality model, and second, our validation approach, which we argue provides more convincing evidence than approaches from literature that the quality model is indeed useful for assessing QVTo transformation quality. Finally, numerous directions for future research and tooling related to QVTo quality are also presented.

2 Preliminaries

2.1 QVT operational mappings

A QVTo transformation (Barendrecht 2010; Nolte 2010; OMG 2011) transforms models conforming to one or more metamodels. The *transformation declaration* specifies the *source* and *target* metamodels. An example of a simple but complete transformation is given in Listing 1 utilizing the ABC metamodel from Fig. 1. The **`main`** function serves as the entry point for the transformation.

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. Mappings have three sections: an optional **`init`** section for object initialization, a **`population`** section for mapping input element fields to the output, and an optional **`end`** section for post-processing. Mappings can also include pre- and post-conditions defined in OCL (OMG 2012), indicated with the **`when`** and **`where`** keywords,

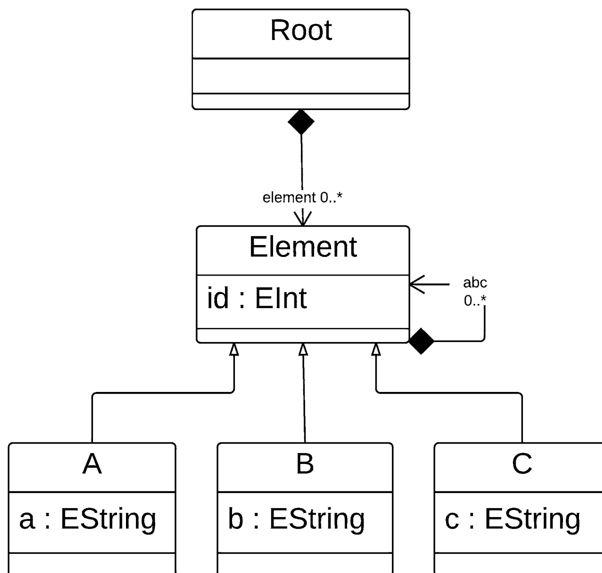

Fig. 1 ABC metamodel

respectively, such as in line 33 of Listing 1. When inside a mapping, the **self** keyword can be used to access the input element and **result** to access the output element.

A **query** is an operation to obtain data from the input object(s), for example a subset of its elements from the model. According to the specification, queries may not have side effects of their parameters. Helpers, declared with keyword **helper**, are like queries but may have side effects. Finally, **constructor** functions can be used to explicitly construct an element from a metamodel. In this paper, we refer to queries, helpers, and constructors as functions. The imperative constructs of QVTo, similar to those available in GPLs, 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.

Listing 1: A simple QVTo transformation using the ABC metamodel

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

As the transformation executes, a *trace* is recorded for every source element transformed by a mapping. By inspecting these traces, developers can see for example the order in which mappings were invoked, giving increased insight into the transformation process (Santiago et al. 2013). Traces also enable *resolving*, a feature very useful for the model transformation domain and difficult to implement in GPLs: 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, the reference can be *resolved* to the earlier transformed object by calling **`resolve`** on the source object. 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 at the end of the transformation.

QVTo also makes available a number of reuse mechanisms, including *composition*, allowing one transformation to explicitly invoke another transformation, and *extension*, where one transformation can build upon another transformation. Functions that are reused by multiple transformations can be placed in *libraries*. Libraries and transformations are both referred to as *modules* in QVTo. Reuse is also supported at the mapping level. A mapping be abstract or can *inherit* from another mapping, causing the other mapping to be called after the **`init`** section of the first mapping has run, or *merge* a number of other mappings, where the specified mappings are to be run after the **`end`** section of the first mapping. Mappings can also *disjunct* a number of mappings, where the first mapping which can accept the input object, typically specified by **`when`** clauses on the mapping, is executed.

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 which store transient information during the execution of the transformation. Global parameters can be declared outside of a transformation using the **`configuration property`** keyword. QVTo also allows calling *black-box* functions written in other languages. Typically, black-boxes are used when a function or transformation is too complex to write in just QVTo and is instead implemented in a language like Java or QVT Relations.

2.2 Software quality

Software quality can be approached from many perspectives (Kitchenham and Pfleeger 1996), some of which we adopt *a priori*. First, we are primarily interested in *internal* quality, which means measuring quality by looking inside the software product (e.g., analyzing source code), rather than measuring the software during execution (e.g., testing) (Ferenc et al. 2014). It follows from this focus on internal quality that the primary "customers" of our investigation are developers rather than testers, which would be more closely targeted when focusing on external quality. We also focus on *direct* measurements of transformation quality, which measure the transformation itself, rather than *indirect* measurements, which measure other artifacts involved in model transformation (e.g., models) to assess transformation quality (van Amstel 2012). We chose these perspectives because they are most likely to provide insights immediately useful to produce higher-quality transformations, our goal with the pragmatist stance.

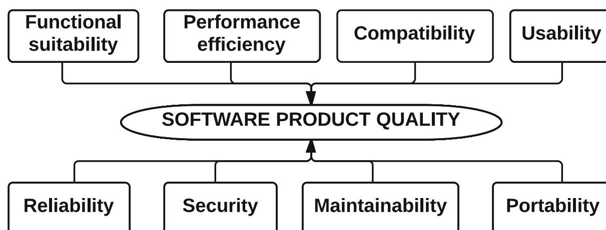

Fig. 2 Quality characteristics for software products in ISO/IEC 25010 (ISO/IEC 25010 2011)

Internal quality is often formalized in a *quality model*. According to ISO/IEC 25010 (ISO/IEC 25010 2011), the standard on system and software quality, a quality model contains high-level quality characteristics, quality attributes, and evaluation procedures. The standard also presents a quality model for software products, containing all characteristics and attributes relevant to software quality in general. The top-level quality characteristics defined there are depicted in Fig. 2. Evaluation methods, however, are not provided in the standard product quality model, so it cannot be used directly to assess code quality. Furthermore, although the quality model is comprehensive, it is not *a priori* clear given the unique range of features and applications of QVTo that it is the optimal model for QVTo in practice. An extensive overview of the standards is available in ISO/IEC 25000 (2014).

Exploring quality in MDE, van Amstel et al. (2010) defined a set of metrics to evaluate model transformation quality, including QVTo transformations. These metrics included size metrics, function complexity metrics, modularity metrics, inheritance metrics, dependency metrics, consistency metrics, input/output metrics, and QVTo-specific metrics. However, because these metrics were proposed based on language features and prior research for GPLs, it is not evident that this set of metrics is useful to assess QVTo quality. Later, van Amstel (2012) provided an evaluation of these metrics with respect to maintainability for ATL and ASF + SDF. There, an average of three experts rated thirteen code samples on seven quality characteristics. The expert ratings were then correlated with metric measurements. For ATL, no significant correlations were found, and for ASF + SDF, while some significant correlations were observed, the high correlations between metrics made it difficult to assess the impact of individual metrics. Identifying individual metrics is particularly important because having too many metrics is one of the reasons metrics programs fail in practice (Hall and Fenton 1997; Moody 2003). Therefore, although quality models have been proposed that are applicable at least in theory to QVTo, there is little empirical evidence that these models represent a useful notion of QVTo quality. We call these approaches starting exclusively from theory *top–down* approaches.

3 Formalizing QVTo quality

To identify the most important aspects for QVTo quality, we follow a *bottom–up* approach based on grounded theory (Seaman 1999). This way, the most *pressing* concerns in QVTo quality are captured, which are most important according to our pragmatist stance. Furthermore, utilizing a bottom–up approach already provides high empirical validity (Moody 2005), particularly valuable in domains where validation is difficult. Our approach consists of two phases. The first is a broad exploratory study to gather substantial support for what affects quality in QVTo. The second phase is formalizing the information gathered into cohesive format which can be used to assess QVTo transformation quality.

3.1 Exploratory study

The exploratory study focuses on gathering *qualitative* data. Since producing high-quality code heavily depends on developer actions, namely how developers write code, qualitative data can provide the richest insights into the complex reasoning processes that developers use (Seaman 1999). Since qualitative methods are sometimes considered “fuzzier” than quantitative methods in computer science (Easterbrook et al. 2008), care was taken in

selecting methods and conducting the study, leveraging in particular the extensive experience with qualitative methods from social science research. To guarantee that the exploratory study produces generalizable data and builds upon previous research and experience, we utilize a triangulation approach: expert interviews, a review of existing material, and introspection.

3.1.1 Expert interviews

Four semi-structured interviews of QVTo experts at ASML were conducted. By gathering data directly from developers, we gain access to potentially years of best practices and learning involved with becoming an experienced programmer. Additionally, by speaking with developers directly, we can quickly obtain insights into which aspects of QVTo are most important for quality today. As with all interview research, the data gathered in this component of the exploratory phase may be very specific to the individuals interviewed. Mitigating this bias while still leveraging rich developer experience is a primary motivation for our triangulation approach.

Table 1 Interview guide used by interviewer during QVTo expert interviews

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?

State-of-the-art techniques for interviews were followed, in particular recommendations from Seaman (1999) and Hove and Anda (2005). Each interview lasted 45–60 min and was audio recorded and later transcribed. A one-page interview guide displayed in Table 1 was used by the interviewer to help direct the interview, containing questions about development processes, typical bugs, QVTo versus other languages, as well as poor practices exhibited by those learning QVTo. To assess the interview format, a mock interview was performed with an independent developer beforehand.

Examples of information gathered during the interviews are that common refactorings include converting imperative-style programming to declarative (since the latter is more readable as well as faster in execution), that a transformation is easiest to work with if its structure mirrors the hierarchy of either the input or output metamodel (but not both), and that unit testing is used extensively by the team, but there does not exist any measure of test coverage.

3.1.2 Existing material review

To leverage the vast amount of knowledge on code quality, an extensive review of existing materials was performed. This review included not only literature, but also other sources such as online forums. The latter are particularly important since the amount of scientific literature on QVTo is still limited and much insight can instead be gained from online sources.

For literature, a *systematic literature review* (Kitchenham et al. 2009) was performed. Per the guidelines for conducting systematic literature reviews (Kitchenham 2004), a review protocol was first developed to reduce researcher bias. The topics included in our protocol were chosen according to our pragmatist stance and are presented along with specific search terms used in Table 2. There, each keyword was combined with a technology term to form the search query (e.g., “QVTo best practices”). “Tooling” is included as a topic because of its importance to writing high-quality code. ATL and Java were chosen for additional specific technologies since ATL has similar imperative constructs to QVTo and both ATL and Java are prominently used for model transformation. 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.

Table 2 Topics, keywords, and technologies used as inclusion criteria during the systematic literature review

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, plug-ins	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 3 Most relevant papers resulting from the systematic literature review

Author	Genres	Most relevant contributions
van Amstel and van den Brand (2011)	Tools	•Three tools for analysis improving maintainability •Metrics •Metamodel coverage analysis •Tools implement for ATL, QVTo
van Amstel et al. (2011)	Best practices	•Performance of MDE language constructs
van Amstel (2012)	Metrics	•Quality metrics for ATL •Validation through correlation with attributes via expert surveys
van Amstel et al. (2012)	Tools	•Visualization of traceability to improve development •Higher-order transformations
Ciancone et al. (2010)	Tools	•Novel unit testing approach for QVTo •Evaluation performed in practice
Fabro and Valduriez (2009)	Best practices	•Automation of transformation development to increase quality
van Dongen (2012)	Tools	•Visualization of QVTo transformations to improve quality
Ergin and Syriani (2013)	Best practices	•Patterns for refactoring to improve quality metrics •Quality framework followed
Gniesser (2012)	Best practices	•"Bad smells" in MDE languages •Refactorings •Validation performed
Guduric et al. (2009)	Best practices	•Recommendations for QVTo code reuse
Kapová et al. (2010)	Metrics	•Transformation maintainability metrics •Metric categories
Kolahdouz-Rahimi et al. (2014)	Metrics	•Evaluation framework for languages for transformation refactoring •Goal-quality-metric framework •Results of factors affecting ISO quality characteristics
Kusel et al. (2013)	Best practices	•Current reuse techniques used in practice and the merits thereof
McQuillan and Power (2009)	Testing	•Transformation test coverage measures
Nguyen (2010)	Tools	•Metrics for QVTo •Tool to measure metrics
Paige and Varró (2012)	Tools	•Guidelines for creating MDE tooling
Planas et al. (2011)	Best practices	•Correctness properties for ATL
Rentschler et al. (2013a)	Tools	•Transformation Analysis tool for QVTo •Validation performed of effect on quality
Rose et al. (2014)	Tools	•Observations from transformation tool contest judgments
Selim et al. (2012b)	Best practices, testing	•Appropriateness of various MDE languages for projects •Case studies •Tool to measure quality of MTM transformations •Expert questionnaire for validation •Analysis of quality goals
Selim et al. (2012a)	Best practices	•Static and dynamic analysis techniques transformations
Stahl and Voelter (2006)	Best practices, tools	•QVTo language usability •MDE tool standards compliance and consistency

Table 3 continued

Author	Genres	Most relevant contributions
Syriani and Gray (2012)	Best practices	•Challenges in quality assessment, including metrics •Need for transformation refactoring and design assistance
Voelter (2009)	Best practices	•Best practices from DSLs •Validation performed
Voelter and Kolb (2006)	Best practices	•Best practices for writing model-to-text transformations •IDE/tool support

The technology terms also served as our sole *exclusion criteria* for literature found, where search queries already returning a great deal of literature for technologies closest to our focus terms (e.g., "QVTo" or "model transformation") were not pursued for the more distant technologies (e.g., "Java"). 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 (onl 2013a). Offline and non-English language sources were not reviewed. The title, publication date, and a list of extracted key points were recorded for each source. Whether the source provided an explicit evaluation of their contributions was also noted to indicate the source's credibility. Rich online sources reviewed were the forum for the Eclipse QVTo implementation (onl 2014d) and the publicly available comments from the transformation tool contest (onl 2013b).

A list of the most relevant papers resulting from the systematic literature review are presented in Table 3. An example of information gathered during this review is that mixing notations (e.g., text, graphical) can reduce understandability. A number of metric sets that have been proposed in model transformation literature (e.g., Kapová et al. 2010; Vignaga 2009) were also encountered.

3.1.3 Introspection

The third component of the exploratory study was learning QVTo by the first author. Here, a series of QVTo tutorials and examples was followed (e.g., Stahl and Voelter 2006). During the learning process, all aspects related to quality were recorded, such as difficulties and realizations about better ways to implement certain functionalities. Since it is important for software engineer interviewers to be knowledgeable of the domain at hand (Hove and Anda 2005), this component was performed before the interview component of the triangulation approach. This component complements the other two by providing the author with firsthand information on novice practices, a cause of low-quality software.

An example of information gathered during introspection is that a novice tends to create large **`init`** sections inside mappings to initialize the output elements, since that is similar to GPL programming, as opposed to using the more concise implementation with a **`population`** section.

3.2 Constructing the quality model

With the qualitative data from exploratory study, we perform what is known in classic theory generation as the *constant comparison method* (Seaman 1999). All *key points*

related to transformation quality are extracted, approximately one sentence per point. Each point is *tagged*, or *coded* (Seaman 1999), with keywords describing why it is relevant for quality, e.g., "conciseness," matching the wording used in the raw data as closely as possible. Then, an iterative approach is applied where similar points are grouped together, while always maintaining traceability links back to the original sources. The set of quality tags is then reduced by combining closely related tags, e.g., "readability" was combined with "understandability."

Until this point, to avoid researcher bias, it was not assumed that a formal quality model would be the result of our investigation of QVTo quality. It became clear, however, that most of our key points and tags could be effectively formalized within a quality model. The quality tags became the *quality goals* of our model, representing the primary quality concerns for QVTo transformations today. We then translated each point into a short phrase. These phrases comprise the *quality properties* of the QVTo quality model. To give an indication of whether the property helps or hurts quality, we also assigned a *directionality* to the property. Directions were chosen so that every property generally increases transformation quality. To accommodate for quality goal trade-offs, however, each quality goal associated with the property was 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 property "More mappings than helpers" with goals "Understandability (+)" and "Performance (−)." We use the term property rather than attribute because attributes generally do not have directionality (ISO/IEC 25010 2011).

Quality properties with insufficient support from our triangulation approach were then discarded. Specifically, properties with fewer than two original sources (for example being mentioned by only a single paper) were removed from the model. Properties which were only proposed in literature with no validation were also removed. This latter step is required by our pragmatist stance to avoid including properties for which there is still no evidence of usefulness in practice. To complete the model, evaluation procedures are added for each property. An elaboration of our approach is given in (Gerpheide 2014).

3.3 Resulting quality model

In Table 4, the QVTo quality model resulting from the exploratory study is presented. It consists of 37 quality properties and four quality goals, namely functionality, understandability, performance, and maintainability. Although the list of properties may seem large, our validation approach (Sect. 3.4) distinguishes their relative importance. The properties have been classified as specific to QVTo, MMT, or neither, in which case they are also applicable to GPLs. The proposed classification is based on the traceability links from our quality model, therefore reflecting why the property is in the model. For example, "Deletion uses trashbin pattern" is classified QVTo specific because it relates to the QVTo implementation used by the interviewed developers. According to our classification scheme, 13 properties are QVTo specific, four are MMT specific, and 20 are applicable to GPLs. The evaluation procedures identified are also of a largely quantitative nature, lending them to being used as automated metrics in future work.

The quality properties have been further organized in Table 4 according to their nature. These "natures" are similar to the transformation quality *objects* presented in the transformation quality framework by Mohagheghi and Dehlen (2007). Two of the properties are presentation related, since they focus on style and are unrelated to transformation behavior. Nine of the properties concern the high-level architecture of a set of transformations modules in a project. Four properties are related to the current QVTo engine

Table 4 QVTo quality model resulting from the exploratory study

Quality property[a]	Quality goals	Evaluation procedure	Applicability
Presentation			
Detailed comments throughout code	U+	Comment/LOC ratio	GPL
Formatting conventions followed	U+	# Violations of a coding standard	GPL
High-level architecture			
Few black-boxes	U+, M+	# Black-boxes	QVTo
Few configuration properties	U+, P−	# Configuration properties	QVTo
Few dependencies on other modules	U+, M+	Module fan-out	GPL
Few input/output models	U+, M+	# Input/output models and metamodels	MMT
Few intermediate properties	U+	# Intermediate properties	QVTo
Low code duplication with other modules	M+	# Instances where at least five lines repeated	GPL
Pre- and post-conditions specified	F+, M+	Presence of formal specification	MMT
Small interfaces to other modules	U+, M+	Function fan-out to other modules	GPL
Small transformation size	U+	Module LOC	GPL
Transformation local			
Confluence satisfied	F+	Proof of confluence	MMT
Few dependencies between functions	U+, M+	Function fan-out within module	GPL
Few **end** sections	U+	# **end** sections	QVTo
Few mapping arguments	P+, U+	# Arguments per mapping	GPL
Few nested **if** statements	U+	Nesting depth	GPL
Few **when** and **where** clauses	U+	# **when** and **where** clauses	QVTo
High test coverage	F+, M+	Test code coverage	GPL
High usage of design patterns	U+	Comparison of patterns used to a pattern catalog	GPL
Inheritance usage matches metamodel	U+	# Abstract classes in common with metamodel	MMT
Interdependent functions near each other	U+, M+	Custom semantic similarity measure	GPL
Little dead code	U+, M+	# Unused LOC	GPL
Little imperative programming	U+, M+	# **forEach** loops	QVTo
Little overloading	U+	# Instances of overloaded functions	GPL
Low code duplication within module	U+, M+	# Instances where at least two lines repeated	GPL
Low syntactic complexity	U+	Syntactic complexity measure (Kolahdouz-Rahimi et al. 2014)	GPL
Minimal reassignment of objects	F+	# Instances when object assigned to multiple sets	QVTo
More mappings than helpers	U+, P−	Ratio # mappings to # helpers	QVTo
More queries than helpers	U+, P+	Ratio # helpers to # queries	QVTo
Short function chains	U+	Length of chains	GPL

Table 4 continued

Quality property[a]	Quality goals	Evaluation procedure	Applicability
Small function size	U+	Function LOC	GPL
Small **init** sections	U+	LOC inside **init** sections	QVTo
Termination checked	F+	Proof of termination	GPL
Implementation-dependent			
Few queries with side effects	U+	# Queries with side effects	QVTo
Deletion uses trashbin pattern	P+	# Instances where trashbin pattern not used	QVTo
Low execution time	P+	Execution speed with typical model	GPL
Mappings only used when tracing needed	U−, P+	# Instances when mapping used but not tracing	QVTo

F, U, P, and M indicate quality goals functionality, understandability, performance, and maintainability, respectively. Properties are organized according to category and then alphabetically

[a] "Function" refers to a mapping, query, or helper. "LOC" is lines of code. "Module" refers to a transformation or a library. "Function chain" refers to a chain of function calls. "Confluence" is a property of declarative languages where output is independent of execution order. *Tracing* internally links input and output elements. Clauses with **when** and **where** specify pre- and post-conditions on mappings. *Intermediate properties* store transient data during execution, and *configuration properties* have global scope

implementation, and therefore may differ between implementations. For example, "Few queries with side effects" is implementation dependent because in the Eclipse QVTo implementation, queries are implemented as helpers, therefore allowing side effects, despite this not being allowed by the QVTo specification. The remaining 22 properties can be considered quality properties local to a transformation, since they are specific to how a single transformation has been written. This categorization helps determine when and where each property could be leveraged to improve transformation quality. For example, if a developer has the opportunity to modify a transformation but is not able to re-architecture his or her project, then the transformation-local properties may be most useful. As another example, if a new version of the QVTo engine is released, it may be necessary to reevaluate the implementation-dependent properties. The high proportion of transformation-local properties is likely a result of our approach, since existing quality models also have high proportions of properties local to a single module or class (e.g., van Amstel et al. 2010; Kapová et al. 2010), and the developers interviewed work also most frequently on modifying single transformations. We do not consider the effect of our approach on the properties a threat to validity, however, since our approach was designed to capture the most pertinent concerns in QVTo quality. Instead, this suggests that the most useful model to assess QVTo quality contains a variety of properties but with an emphasis on transformation-local concerns.

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. This is expected, however, due to our focus on internal quality and because understandability and maintainability are more complex goals to describe than performance or functionality, therefore requiring more properties. Furthermore, although performance may be most related to external quality, it is still incorporated in our model because it was clear from our exploratory study that any description of QVTo quality without a consideration for performance would be incomplete.

Some properties in the model are of particular interest. For example, "Little imperative programming" may seem unintuitive, since QVTo was specifically designed to include

imperative constructs. However, our exploratory study suggests that although these constructs are indeed useful for creating more complex transformations, where the alternative would be using a black-box containing Java code, they should be minimized where possible. In the experience of one of the developers interviewed, “80 % of transformations can be written in a purely declarative fashion, making them much easier to understand and work with later on.” Property “Mappings only used when tracing needed” is also notable, since although mappings are a core concept of QVTo and avoiding them decreases understandability, it emerged as more important to improve performance, therefore serving as an example where a quality model not considering execution performance may be less useful for developers.

That the model was built bottom–up can also be seen. For example, the property “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 properties included in the model are there because they emerged as relevant to quality during the triangulation approach.

3.3.1 Model scope

Because our approach was not restricted to specific types of QVTo transformations, we assert that our model is relevant to all strata of the model transformation taxonomy proposed by Mens and Van Gorp (2006) that are applicable to QVTo. Specifically, the quality model can be used to assess quality of QVTo transformations with one or multiple-source and/or target models. Since QVTo supports source and target models having different metamodels, the quality model can be used on both endogenous and exogenous QVTo transformations. Similarly, the model can be used to assess both horizontal transformations (i.e., performing refinements or abstractions) and vertical transformations (i.e., which do not change the level of abstraction). The experts interviewed also worked regularly with all of these transformation types. However, we do not claim that the quality model is applicable to MMT languages other than QVTo without further study.

We also place our model in the context of the transformation *intents* proposed by Amrani et al. (2012). Since QVTo could be theoretically used for each of the 17 most common intents identified there, the quality model proposed here is also applicable to transformations with those intents. Of the 17 intents, however, the experts interviewed had less experience in working with transformations addressing the “approximation” and “model generation” intents. Due to the importance of the interview data in constructing our quality model, we therefore consider this a threat to validity when using our quality model to describe QVTo transformations with those intents.

3.3.2 Similarities to other models

The model most closely related to our QVTo quality model is the set of QVTo quality metrics proposed by van Amstel et al. (2010), introduced in Sect. 2.2. Of our 37 quality properties, approximately one-third have corresponding metrics in their metric set (which coincidentally also contained 37 entries). Some of the similar metrics are nearly identical (e.g., metric “# Overloaded mappings” and our property “Little overloading”), while others are similar but not equivalent (e.g., metric “# Trace resolution calls” and property “Mappings only used when tracing needed”). These similarities appear in each of the

metric categories identified by van Amstel et al. (2010), suggesting that our synthesis approach also succeeds in covering a similar range of concerns as a top–down approach.

The similar metrics, however, comprise the more straightforward properties from our model (e.g., "Few input/output models"). Our more complex properties (e.g., "Deletion uses trashbin pattern" and "Interdependent functions near each other"), on the other hand, do not have counterparts in the metric set. In some cases, a metric may seem similar to a property, but as a result of the bottom–up approach, the property is substantially more nuanced. For example, while the set from van Amstel et al. (2010) includes the metric "# Abstract mappings," our model includes the property "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. The properties in our quality model are also in general more concise than the metrics. For example, five individual metrics are proposed for number of unused mappings, helpers, parameters, and local variables, whereas our quality model simply includes "Little dead code," which in our model is evaluated by measuring lines of unused code. Conciseness is valuable since it again reduces the risk of overwhelming practitioners with too many metrics, as mentioned in Sect. 2.2.

Notably, a number of properties identified as important during our exploratory study are not present in van Amstel et al. (2010). 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 van Amstel et al. (2010). Our model also includes more QVTo-specific properties (e.g., "Few **end** sections"). These may be excluded from van Amstel et al. (2010), however, because there are simply too many language-specific constructs to add a property for each construct. This difficulty to distinguish the most important features of new domain is in our opinion a fundamental problem with top–down approaches.

In Van Amstel's quality model for MMT languages in general (van Amstel 2012)(cf. in Sect. 2.2), a number of additional quality attributes are present. Some are similar to our quality model, e.g., "Depth of inheritance tree" proposed for ATL which is closer to our "Inheritance usage matches a metamodel" property. However, due to the top–down approach, van Amstel's metrics (van Amstel 2012) describe relatively shallow properties compared to the rich data incorporated in our quality properties.

Kapová et al. (2010) presented a set of maintainability metrics for QVTr. There a combination of "automated" and "manual" metrics was proposed. The automated metrics, like those from van Amstel et al. (2010) and van Amstel (2012), are simple, including "# Local variables" and "# **when** predicates" (the latter of which corresponds directly with our property "Few **when** and **where** clauses"). The manual metrics, however, contained "Similarity of relations,"[1] "# 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" property, 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 authors), rather than contributing to understandability like our property. Because this property was voiced as important by multiple interviewees for transformation readability and understandability, its explicit link to understandability is a valuable addition to our model.

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

Therefore, while there is some overlap between top–down models and our quality properties, we advocate the use of bottom–up approaches in future quality research to obtain the data most relevant in practice.

3.3.3 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 demonstrated either by using the standard models directly or by showing traceability links between the standard model and a tailored model. We demonstrate conformance using the latter method, showing the links from the software product quality model to 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, 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 properties can each be mapped to a quality attribute in the standard quality model by removing the directionality.

3.3.4 Best practices

As a result of not restricting the exploratory study to building a quality model from the beginning, some data from the study can be formalized better as best practices than as quantifiable quality properties. Here, we present a selection of best practices gleaned in particular from the expert interviews, constructed by performing the constant comparison approach using just the interview data. Although not validated in Sect. 3.4 and highly specific to the experts' experiences, these can be used as a starting point for other QVTo practitioners to formalize their own best practices. They also provide an example of the rich data gathered using a bottom–up approach.

First, navigation over models should be separated from mappings as much as possible, ideally by creating a query library for each metamodel. This way the functions used to traverse a given metamodel can easily be reused by transformations. Second, **`init`** sections are appropriate only when objects need to be *explicitly* constructed, e.g., selecting the concrete type of an object from the abstract type. Otherwise, developers should leverage the implicit initialization done by the **`population`** section. Third, when multiple objects must be generated and assigned from one input object, a **`constructor`** should be used instead of the assignment operator since then it is guaranteed that a new object is created rather than resolving to a previously created object. Finally, if performing incremental transformations with large input models, the traces can be utilized to avoid unnecessary regeneration of the target model, improving performance.

3.3.5 Difficulties

In addition to best practices, many current issues were identified which can serve as areas for future work in QVTo research. First, lack of documentation as well as lack of (non-

buggy) tooling were both identified as major problems. Eclipse editor support could be improved, for instance preventing accidental reassignment of objects. There are also many limitations in the debugger, such as poor support for chains of transformations, so developers often default to using `log` statements instead. Testing frameworks are still considered immature, and in particular no measure of test coverage exists. Finally, there are also inconsistencies between the QVTo specification and Eclipse implementation used by many developers, for example side effects being allowed in queries and not supporting deep recursive calls.

3.4 Validating the quality model

Although the bottom–up approach guarantees initial support for each property, validation is still required for confirmation and to show generalizability. Before introducing our validation approach, however, we discuss the drawbacks of the approach used in the most closely related work (e.g., van Amstel 2012; Lehrig 2012). There, transformation quality models were validated by conducting a survey where experts rated code samples on quality goals, which were later correlated with the measured quality metrics. In addition to the conclusions being drawn from this validation approach being generally weak, it is clearest to see its drawbacks by simulating it on our quality model.

We observe that in this method, experts are required to make snap judgments based on a visual sample of the code. This method therefore first strongly biases judgments toward visual properties, such as properties related to readability. Therefore, properties related to other quality goals (e.g., "Termination checked") could incorrectly receive lower correlations. Second, properties which cannot be deduced visually at all (e.g., "Low execution time") or cannot be deduced *quickly* (e.g., "Few dependencies to other modules") would likely go unnoticed by the reviewer. Furthermore, there are simply so many properties that it is not possible to control for each individually, making it impossible to evaluate the relative importance of the properties; to do so would require preparation of an extremely large set of code samples for experts to review, which is infeasible within reasonable time constraints. This effect would in particular lead to lower correlations for less-common properties (e.g., "Small `init` sections"). Hence, a large portion of our quality model would likely go unvalidated if we were to use this method, regardless of whether the properties are important for QVTo transformation quality.

We propose a different validation approach for our quality model. With our assumption that developers themselves are one of the best resources to determine what high-quality code is, we validate our quality model by performing a survey of developer perceptions of each quality property. Using perceptions is based on the pragmatist stance, directly leveraging practitioner experience. By addressing each property individually, we assess the model on a very fine-grained level.

The validation survey consisted of an introduction to the research, instructions, a field for the developer to describe their experience with QVTo, and finally, a question regarding each quality property. Definitions of each quality goal as they relate to transformation quality were also clearly presented (see Gerpheide (2014) for the complete text). Each property question was a compound Likert-style (Johns 2010) question asking about the relationship between the property and the four quality goals. An example question is shown in Fig. 3. All property questions were worded similarly, since although acquiescence bias (where respondents passively indicate responses which agree with the question) is a drawback of Likert scales, it is more important to have this bias consistently throughout the survey (Barnette 2000). After each question was a field for comments and for less-known

In your opinion, what effect does Few Intermediate Properties have on a transformation?
(measured by: # intermediate properties)

	Strongly decreases	Decreases	Somewhat decreases	Has little/no effect	Somewhat increases	Increases	Strongly increases
Functionality	○	○	○	○	○	○	○
Understandability	○	○	○	○	○	○	○
Performance	○	○	○	○	○	○	○
Maintainability	○	○	○	○	○	○	○

Fig. 3 Example question from validation survey

QVTo or programming terms, a short explanation was provided. The survey instructions stressed that the developer should answer the questions in terms of typical QVTo transformations and should reflect their own experiences. A mock survey was carried out with an independent developer to test the survey clarity and completion time. The survey was distributed online by posting on the QVTo forum and the QVTo developer mailing list, and the link was sent directly to the ASML QVTo developers.

Fifteen respondents filled out the survey, including the original four experts interviewed during the exploratory study and the two remaining ASML QVTo developers. The remaining QVTo experts included three from industry, five graduate-level students, as well as one of the four primary committers of the Eclipse QVTo implementation. Respondents had between 1 and 9 years of experience working with QVTo.

The survey results are summarized in Table 5. To measure overall agreement, we calculated Kendall's coefficient of concordance W (Field 2005), a measure of respondent agreement where 1 implies perfect agreement. For our response set, $W = 0.42$ (p value $< .01$),[2] which can represent weak overall concordance (Vidmar and Rode 2007), implying that at least some properties may have high disagreement. To better interpret the coefficient, we examined the interquartile ranges for each quality property/goal pair, where an interquartile range of one represents 50 % of developers having a responses which neighbor each other on the Likert scale. We find that 80 % of pairs had an interquartile range of one or less, which we interpret as high agreement for most properties.

In general, presentation-related properties had the highest agreement. The GPL-applicable properties also had higher agreement in general than the MMT- or QVTo-specific properties. The pair which was least agreed-upon 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.

Comparing the ASML team to the public, $W = 0.44$ and $W = 0.32$, respectively, so there is more agreement within the ASML team than among 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. The median answers for "Termination checked" and "Confluence satisfied" were also considerably higher for the ASML team than for the public. This we explain by differences in academic background (two ASML team members have PhDs in formal computer science topics, whereas members of the public did not have formal computer science backgrounds).

[2] Calculated in R with correction for ties using the **`irr`** package command **`kendall(df,correct=TRUE)`**

Table 5 Validation results

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

To see the relationship between individual survey respondent answers, we calculated the pairwise correlations between responses, shown in Fig. 4. Respondents 1–6 were the members of the ASML team. Correlations were calculated using the nonparametric statistic Kendall's τ as well as the parametric Spearman's ρ, giving equivalent correlations with both statistics, thereby increasing the confidence in the coefficient values. It is clear that some respondents had much more similar responses than others, for example respondent 12 correlated highly with 14 and 15. Looking deeper, this high correlation came largely from GPL-applicable properties, and these three respondents in fact 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.

Next, we consider which property/goal pairs are perceived as having the strongest impact on quality by comparing median responses. The pair with the highest median was "Low execution time"/Performance followed by properties "Little dead code," "High test coverage," and "Few black-boxes," 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,"[3] a trade-off is clearly acknowledged by the respondents where it decreases understandability but increases performance.

Finally, and arguably most importantly, we define two criteria which must be met by a property/goal pair before we consider it validated by the survey. First, the median answer must be at least "Somewhat increases." Second, at least 75 % of the responses must have been at least "Has little/no effect." The results which satisfied these criteria are shaded gray in Table 5. Of our original 37 properties, 26 are therefore validated for being important for at least one quality goal. Of those validated, nine were considered MMT- or QVTo-specific.

A sensitivity analysis was also performed where each respondent was removed one at a time and the two criteria were rechecked. The three pairs which did not satisfy the criteria in every round are struck through in Table 4, suggesting that these pairs in particular may benefit from additional validation. There are also four pairs which satisfied the criteria only when one respondent was removed. These pairs are displayed in boxes. Since these pairs also appear highly sensitive to the specific set of respondents used and because these pairs are all MMT or QVTo specific, they also make interesting candidates for additional validation techniques.

3.4.1 Threats to model validity

Our approach is not without limitations. First, the quality model is not a complete picture of QVTo quality, since it only contains the properties representing the most important issues at the time of our research. Similarly, although we argue these properties are indeed useful to practitioners, they may not be independent, requiring future work such as factor analysis to pinpoint any independent or underlying attributes. Second, 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. Third, 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

[3] In the "trashbin pattern" objects are assigned to a parent object before deletion.

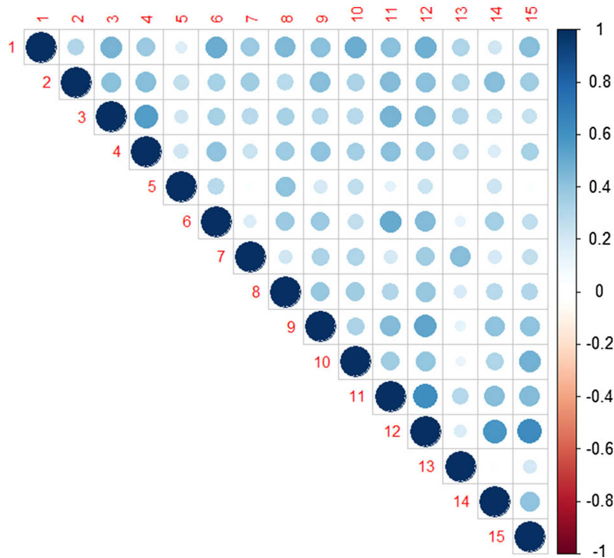

Fig. 4 Pairwise correlations of responses between respondents calculated with Spearman's ρ

take the survey. Finally, although our validation method is more likely than other methods to lead to fine-grained results already useful to practitioners, both our quality model and validation results are susceptible to overfitting due to small sample sizes. For example, ASML may have a particular development style which is not shared by other QVTo developers, providing non-representative interview data. Also, because QVTo is still a young language, many of the QVTo developers today are the same as those creating the QVTo implementations and tooling, and therefore may have different quality needs than future QVTo developers. This overfitting, in fact a common concession in the pragmatist stance (Easterbrook et al. 2008), is the primary motivation of using a triangulation approach. Nonetheless, conducting additional interviews with developers from other domains and backgrounds is an important step for future validation.

4 Leveraging the model in developer tooling

The QVTo quality model presented in Sect. 3 primarily addresses our first research question, namely how do we assess QVTo quality? However, the quality model presented there provides minimal practical value without complementary tooling. Fortunately, we can leverage the QVTo quality model to investigate our second research question: how do we *develop* higher-quality QVTo transformations? Specifically, since developers create the transformations, how can we distill the QVTo quality model in tooling which helps developers create higher-quality model transformations?

The need for tooling is further stressed in literature. Rahim and Whittle (2015) note 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 (2012) also identified the distinct lack of tooling for MDE as one of the biggest current 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.

The attributes from our quality model support many directions for tooling, which we classify here in informal categories. Given their benefit to effectively producing code, tools in many of these categories are already part of standard development environments for many GPLs.

First there are tools which solely measure and report metrics. These could for instance measure the best-validated attributes from the QVTo quality model over time and/or across projects and could be implemented as either stand-alone assessment tools or integrated developer tooling. Another category is visualization tools. In additional to visualizing metrics, they can also visualize structure of a piece of code, which can again be performed statically, between projects, or over time. An example is the transformation analysis tool by Rentschler et al. (2013b), a visualization tool to help developers reduce intramodule dependencies and increase development speed by displaying a visual of dependencies between transformation mappings. Similar visual treatment could be provided for some of the high-level architecture or transformation-local metrics from the QVTo quality model, 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. Other examples of a visualization tools are the TraceVis tool presented by van Amstel for visualizing the traceability links in model transformations (van Amstel et al. 2012) and the MDE code generation environment tool developed by Guana and Stroulia (2014). The trace analysis tool discussed by Santiago et al. (2013) can also be considered a visualization tool. A third category is analysis tools. Such tools could check formal behavioral conditions such as “Termination checked” and “Confluence satisfied,” both which have long been attractive components of automated software development tooling (Ramamoorthy and Ho 1975). For model transformations, little tooling has been developed for these properties, though much research exists on proving them (e.g., Ehrig et al. 2005; Orejas et al. 2009). Neither of these properties, however, were validated with our survey, so they make less attractive tool candidates at this time. An analysis tool addressing a well-validated attribute would be a profiling tool, namely addressing attribute “Low execution time.” Finally, there are process-oriented tools which are most closely related to integrated developer tooling. One such tool is a code coverage tool which measures and displays test coverage of the transformations, thereby addressing attribute “High test coverage attribute,” which was validated to be important for both functionality and maintainability. Another candidate is refactoring support, already available for many GPLs directly from the IDE. Generally speaking, bad smell detection (e.g., Fowler 1999) is a precursor for refactoring tooling. Refactoring and design patterns, for GPLs as well as model transformations, have received much attention in research (Fowler 1999; Ergin and Syriani 2013; Syriani and Gray 2012). In our model, refactoring support could address “Formatting conventions followed,” “Few queries with side effects,” “Deletion uses trashbin pattern,” “High usage of design patterns,” and the attributes concerning code duplication.

Of the options presented above, we have chosen to implement the test coverage tool. The “High test coverage” attribute was well validated for both functionality and maintainability, 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 provided the clearest benefit in improving the identified transformation quality goals. Such a tool thus provides an actual

practical application of the quality model for QVTo practitioners. We use the terms *code coverage* and *test coverage* interchangeably, since both are commonly used in industry to refer to test coverage tools. 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."

4.1 Tool development approach

To support replicating our development approach for the development of other tools, such as for those identified in Sect. 4 or for additional code coverage tools targeting other transformation languages, we make explicit our context, assumptions, and when and why design decisions were made. Hall and Fenton (1997) provide a number of general recommendations for implementing successful metrics programs. Since many forms of developer tooling require presenting useful metrics about code to developers, many of these recommendations apply to our tool development, including 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 (2012) also noted in their experience of developing MDE tooling for 10 years that in general rich graphical views of models are more useful than editing tools. As we are targeting tools for the same domain, we also take these recommendations into account when designing our tool.

Already leveraging two suggestions from Hall and Fenton (1997), we follow an iterative development approach with significant developer involvement. This iterative approach lends itself to the pragmatist stance, since every iteration incorporates developer feedback to maximize the tool's usefulness according to their context and needs. The steps of our development process are enumerated in Table 6.

4.2 Tool design

In particular when leveraging the pragmatist stance, it is essential that a tool is appropriate for the technical environment in which it will be used. Our tool will initially be used in the ASML team using QVTo (the developers in this team are described in more detail during tool validation in Sect. 4.4). The ASML team tests their transformations using JUnit unit tests in Eclipse. There, test cases are created which specify one or more input models, execute the specified transformation, and compare the transformation output model(s) to an expected output model. A depiction of this process of executing a single test case is given

Table 6 Iterative development process used to develop the QVTo quality tool

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

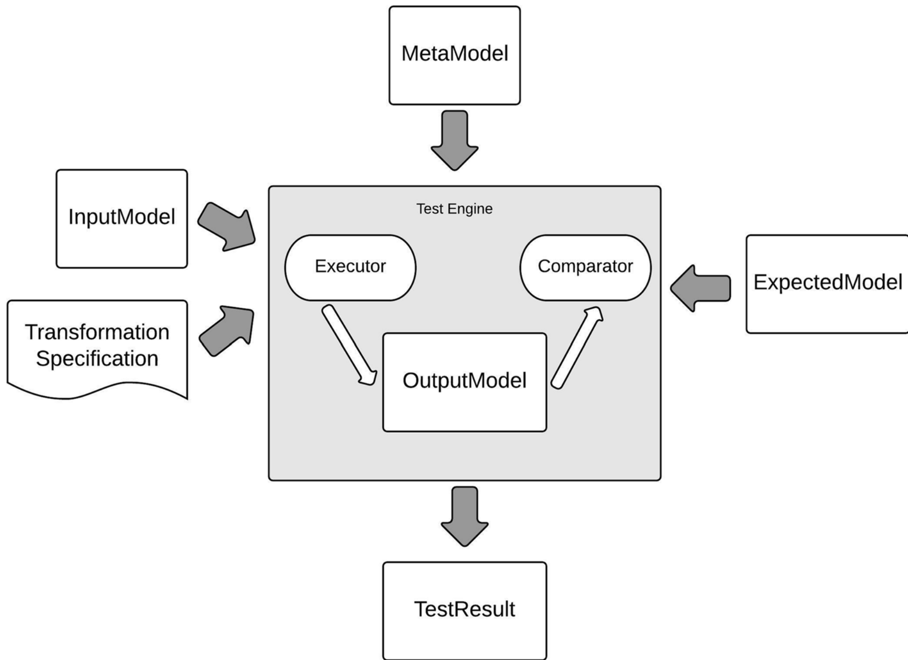

Fig. 5 Model transformation unit test case execution (Lin et al. 2005)

in Fig. 5. According to survey of model transformation verification by Rahim and Whittle (2015) and the description by Lin et al. (2005) 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 development teams using QVTo.

4.2.1 Tool requirements

First, according to our development process (presented in Table 6), a proof of concept in the form of a small Eclipse plug-in was developed to exemplify 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 for Java, onl 2014c). Per step two in our process, this proof of concept was demonstrated to five ASML QVTo developers in a 1-h session to get feedback. From this feedback, the following list of initial requirements was formulated, step three of our development process. Many of the requirements match 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 Sect. 4.2.2.)
- 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 entirely fulfilled in the prototype because the Eclipse QVTo implementation does not expose a straightforward method to easily obtain the total number of unvisited expressions or statements in a transformation. It does, however, expose ways to obtain the unvisited functions. Therefore, red highlighting was added to unvisited mappings, helpers, and queries, but not other unvisited code residing within a visited function.

The prototype was used by developers for 3 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 plug-in “solved their needs” for code coverage. 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.

4.2.2 Coverage criteria

Because test coverage criteria have not yet been defined specifically for QVTo, we define coverage criteria for QVTo here. A number of coverage criteria have been proposed for MMTs in general. These criteria fall into three categories: input metamodel coverage, transformation coverage, and generated code coverage (McQuillan and Power 2009). 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. Transformation coverage variations 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 (McQuillan and Power 2009). Due to the pragmatist stance, in our tool, we focus on

transformation coverage. This is because the test infrastructure used by the ASML 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 in assessing their test coverage.

Because QVTo offers language constructs that are not available in other languages, we define transformation coverage criteria that are specific for QVTo. The following coverage criteria were identified initially: mapping coverage, helper coverage, query coverage, constructor coverage, and statement coverage. In the prototype, all criteria were calculated except query coverage and statement coverage due to limitations in the Eclipse implementation of QVTo. Specifically, statement coverage was not calculated because the concrete syntax of the language does not define a construct for what QVTo developers consider a "statement," namely any expression ending with a semicolon. Since expressions are explicitly defined in the concrete syntax, statement coverage was replaced with expression coverage as a coverage criterion. Query coverage was not calculated because queries are not distinguished from helpers in the concrete syntax and the interpreter treats both constructs as helpers. In the prototype feedback, developers also identified that *total* function coverage, in which mapping, helper, and constructor coverage are aggregated, would be useful to get a quick overview of the coverage of a module. Therefore, five final transformation coverage criteria are identified as useful for QVTo model transformations, presented in Table 7.

For expression coverage, it is essential to understand that expressions are nested. Therefore, the percentage representing the total number of visited expressions divided by the total number of unvisited expressions may not seem accurate at first glance. To illustrate the nested nature of expressions, a very simplistic sample expression is given in Fig. 6. In the figure, a total of five expressions are shown. If the word "World!" were not evaluated during test execution, this code would yield an expression coverage of 4/5. Since it is hard for everyday users to see how that specific number was calculated without knowing the exact expression structure, the coverage value could be perceived as unintuitive. Therefore, two options were investigated for calculating and showing expression

Table 7 Test coverage criteria identified for QVTo model transformations

QVTo coverage criteria
– Mapping coverage
– Helper coverage
– Constructor coverage
– Total function coverage
– Expression coverage

Fig. 6 Nested structure of QVTo expressions shown to three levels deep

coverage: one option using all expressions and one using coverage calculated using only with *leaf* expressions, i.e., expressions with no subexpressions. To assess each option, the expression coverage percentages on a number of test sets were compared. Because of the large number of expressions in a transformation, however, the percentages reported by each method are comparable. For example, the leaf expression coverage for the code samples included in Fig. 7 (including two small functions not shown) was calculated as 6 out of 10 expressions = 60 % and the total expression coverage was 16 out of 29 expressions = 57 %. However, the two methods produce drastically different expression coverage visuals (required by REQ3). The difference is visible in Fig. 7.

Both options were demonstrated to the ASML developers on real test sets to determine which was most useful. There, developers voiced a clear preference for the total expression coverage, citing that 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, making it even "barely usable." Good visualizations are also essential to address the quality attribute for "Little dead code," since they can make dead code make it much easier to identify untouched spots in the code. Furthermore, the developers felt that the percentages reported were equivalent for their purposes. Therefore, total expression coverage was used for the expression coverage criterion as well as in the expression coverage visual.

(a)

```
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)

```
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);
}
```

Fig. 7 Expression coverage visualization options. (a) Leaf expression coverage, (b) Total expression coverage

4.3 Tool implementation

The QVTo code coverage tool is implemented as a fully integrated plug-in for Eclipse. A high-level depiction of the architecture is provided in Fig. 8. To satisfy REQ6, specifying that the tool should make use of existing JUnit launch configurations, the coverage tool contributes an extended Eclipse launch configuration called the *QVTo coverage launch configuration* which can be run on existing JUnit test configurations. This implementation also satisfies REQ1, since directly reusing the existing launch configurations automatically provides support for running any test combinations. The tool collects coverage data by instrumenting the language interpreter's **`Visitor`** class. Overall, there are 18 Java classes included in our implementation organized in six packages, totaling approximately 1700 lines of code. In accordance with Eclipse best practices, the QVTo coverage functionality is divided into a number of distinct plug-ins which can then be easily distributed via an Eclipse update site. Logging functionality was also added to the tool to record usage data to later be used during tool evaluation.

In implementing the tool, a number of code patches were submitted to the Eclipse QVTo core engine (viewable at onl 2014i, j, k). 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. Without these patches, instrumentation of the Eclipse QVTo implementation by third-party plug-ins was not possible. These patches have already been incorporated into the Eclipse Luna version, released in June 2014. An elaboration on the technical implementation of the tool and the QVTo core patches is provided in Gerpheide (2014).

The tool user is exposed to the frontend architecture of the tool, namely the views inside Eclipse, shown in Fig. 9. On the bottom the *QVTo Coverage View*. The left column of the coverage view shows a hierarchical view of transformations that were invoked during the test run. The coverage view cells are colored according to the percentage calculated for that criterion, by default set at 30 and 90 % but customizable in the Eclipse preference pane.

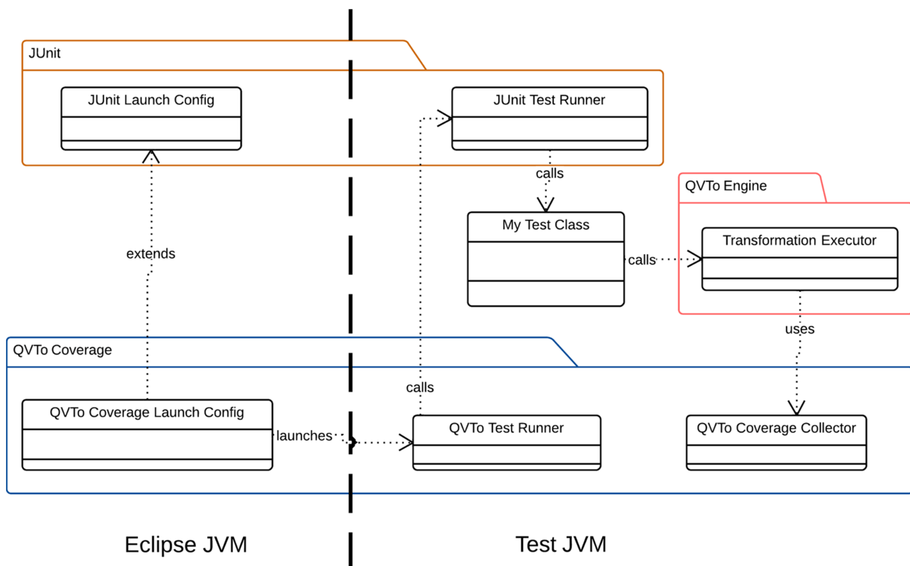

Fig. 8 High-level architecture of QVTo Coverage Plug-in

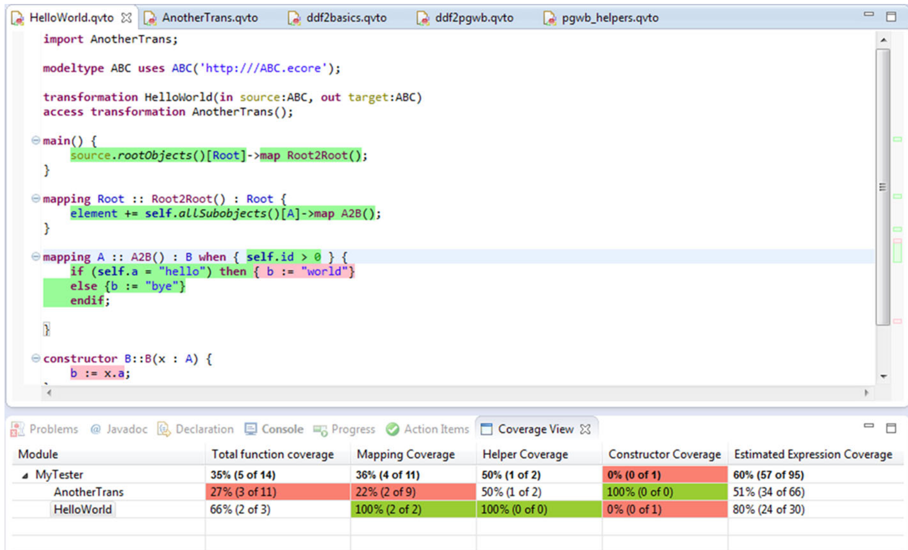

Fig. 9 Coverage and editor views after a test run

When the user double-clicks on the module name, the file is opened in the regular Eclipse editor view with the coverage overlay showing the visited expressions in green and the unvisited expressions in red. 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.

To maximize the usefulness and impact of the tool, it has been made open source under the Eclipse Public License (onl 2014h) and placed online so that anyone can use and modify the tool for free (onl 2014e). Consistent with discussions with the Eclipse QVTo maintainers, it is hoped that our tool becomes incorporated into the Eclipse project by the Mars version release in 2015.

4.4 Tool evaluation

Because our development approach utilized an iterative approach, some initial validity is already provided for the tool in its ability to help developers create higher-quality transformations. However, additional evaluation is desirable. Kitchenham (1996) presents a methodology for evaluating software engineering tools. There, they identified two main categories of evaluation methods: quantitative methods which establish measurable effects of using a tool, and qualitative methods which establish the appropriateness of a tool in a giving setting. We evaluate the tool here using both quantitative and qualitative methods.

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 ASML team used the tool for a period of time on their current QVTo projects after which practitioner feedback is gathered. Practitioner feedback is essential for the pragmatist stance because practitioners must be the ultimate judge of how useful the tool is in practice. This feedback is gathered through semi-structured group interviews because group interviews allow 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. The usage data collected during the tool usage period also contributes to the qualitative analysis, since it can be used to improve the quality of developer's feedback. Performing the qualitative analysis with additional teams outside ASML is identified as an area for future work.

We then perform a quantitative case study in the form of a timing analysis to assess the tool's performance. Performance is a critical factor in the tool's usability, directly affecting its usefulness to developers in a practical setting. Since this analysis provides additional data 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 qualitative evaluation.

4.4.1 Qualitative evaluation results

The usage period lasted a total of 7 weeks. During that period, the coverage tool was used on every QVTo test run by the ASML team. Specifically, usage data reported a total of 98 test runs (either a test set or test suite) performed with the coverage during which 16,714 unit tests were executed.

Five ASML developers participated in the evaluation interview. The interview lasted approximately 45 min and followed the interview guide presented in Table 8. 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* (Beck et al. 2001) to describe how to complete a specific feature or task of an agile sprint. In preparing the user story, the developer ran the entire test suite and recorded which additional tests would need to be written before making the code changes for the feature. The developers also voiced that with the highlighting they could easily see which code may be dead code, providing support that the tool indeed addresses the second quality attribute, “Little dead code,” as well. However, the developers also noted that it is difficult to tell whether code is dead because of the transformation structure or whether it is dead because of the metamodel structure (i.e.,

Table 8 Interview guide used for tool evaluation interview

QVTo code coverage tool interview guide
– 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?

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 (van Amstel et al. 2012).

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." 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 followed by the total function coverage. Although the remaining coverage criteria were considered less useful, the developers still asserted that it was nice to have the extra information there and at least no hindrance to tool usage. One of the developers did note that additional types of coverage used by GPL coverage tools, such as proper statement coverage and branch coverage, would be nice to add. The developers also expressed that it would be useful if the coverage thresholds on which the cell coloring is based could be specified on a per module basis as well, since some transformations, and in particular libraries used by many transformations, are considered to be more critical than others and should therefore be held to higher coverage standards.

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." Based on the feedback above, we consider the tool to successfully increase quality in the context of the ASML team.

4.4.2 Quantitative evaluation results

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 ASML team when using the QVTo coverage launcher versus the standard JUnit launcher. The first test set was comprised of 28 individual tests, touching seven QVTo modules containing (according to the QVTo coverage tool) a total of 3959 expressions of which 2767 were touched. The second test set contained 45 tests with four QVTo modules and 3023 total expressions of which 1464 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 computing environment.

From the timing results, we found that the test set run takes from 16 to 20 % more time when using the coverage tool, varying depending on the test case. By comparison, the EclEmma tool (onl 2014c) commonly used for measuring Java code comparisons is expected to induce approximately 10 % overhead (onl 2014b), though this number, like ours, is highly test case dependent, ranging from 5 to 30 % in normal projects (onl 2014f). Therefore, our average overhead is within the same range. Furthermore, since the overhead incurred with using coverage amounts to only a couple seconds for an entire 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. Developers confirmed that the tool exhibited no noticeable difference in time performance. More details of the timing analysis results can be found in Gerpheide (2014).

4.4.3 Threats to tool validity

One threat to tool validity is the potential inapplicability to other development teams. Although our research suggests that the tool environment used at ASML is a common one, it is likely that there are teams developing in QVTo that require the tool to be adapted to their development setup before it can be used. Secondly, the tool qualitative evaluation should be made more thorough by extending the tool usage period, incorporating more iterations of development, and performing the evaluation with additional teams. A final threat to validity is, like in the survey used to evaluate the quality model in Sect. 3.4, that developers may speak more positively about the tool during qualitative evaluation as to not offend the interviewer. To combat this, it is recommended in future work that evaluation interviews be conducted by an independent party.

5 Related work

This paper directly extends the research on the QVTo quality model presented by Gerpheide et al. (2014a, (2014b). It does this primarily by leveraging the QVTo quality model to identify and develop tooling that directly helps practitioners develop high-quality QVTo transformations. In addition to presenting tooling, an elaboration on the approach to building the quality model, additional insights in evaluating the model, and a more in-depth look at the properties of the quality model itself were also provided in this paper in comparison to Gerpheide et al. (2014a, (2014b).

This work is also related to much work on software quality. However, since the amount of work published on software quality is vast, we only mention the work most closely related to this research. The work discussed here was also incorporated in the exploratory study used to construct our quality model. Ferenc et al. (2014) provide an introduction to software quality models with respect to maintainability. Syriani and Gray (2012) enumerate the challenges in model transformation quality and propose two directions for research: 1) cataloging transformation design patterns and 2) identifying quality criteria for transformations, including quantitative metrics. Therefore, Syriani and Gray (2012) serve as motivation for our research, and we directly build on the second research area they identified.

Like our research, other research has also addressed model transformation quality. However, we classify these as top–down approaches because the authors construct their notions of quality exclusively from theory and related work. In addition to the work described in Sect. 2 and MMT quality models discussed in Sect. 3.3.2, Vignaga (2009) proposed a set of ATL metrics and Kapová et al. (2010) defined metrics for the declarative transformation language QVTr, though in neither case was an empirical validation performed. These approaches heavily influenced our research, first by showing that new metrics can be defined for model transformations, and second, these works motivated us to pursue a bottom–up approach to build a quality model useful for practitioners. Using a

bottom–up approach is further motivated by Hall and Fenton (1997) recommendations for creating successful metrics programs: among their eleven recommendations were transparency, usefulness, developer participation, feedback, and a goal-oriented approach; all of which were heavily incorporated in designing our approach.

Moody (2005) provides a rich overview of techniques to evaluate quality models, pointing out that a surprisingly low proportion of quality models proposed in literature are validated. Each technique is related explicitly to philosophical stances. A demonstration of one of these evaluation techniques is provided by Moody in his assessment of data model quality (Moody 2003). There, a set of data model metrics was proposed. To evaluate the metrics, action research was performed where the metric set was applied in multiple development projects over the course of 2 years, refining the metrics iteratively. Of the original 29, only five remained at the final iteration. Metrics were removed primarily because it was unclear to practitioners how they were useful for quality assessment. The result was therefore a concise set of metrics which were validated to be useful for assessing data model quality in practice. Although action research has not been used to validate our quality model, we consider it a promising direction for future research.

Rahim and Whittle surveyed current work in model transformation verification (Rahim and Whittle 2015), 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). 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 often language dependent.

6 Conclusion

In this paper, a quality model was presented in detail for QVTo transformations. Our bottom–up approach combined a systematic literature review, expert interviews, and introspection, improving upon previous work in software quality models by identifying the aspects *most* relevant for quality assessment in practice. Therefore, the QVTo quality model presented here captures the aspects most useful for QVTo practitioners today. The quality model was then validated by conducting a survey of developer perceptions on how each property relates to QVTo transformation quality. This validation approach also improves upon previous work by leveraging expert opinions to provide fine-grained validation data for every quality property. Of the 37 quality properties included in our model, 26 were considered validated as being important for quality of QVTo transformations. Of the

validated properties, nine are specific to MMT or QVTo, showing that quality models created for other languages will not cover some of the most important properties to assess QVTo quality. Moreover, although the quality model presented here can only provide guidance for quality models targeting other languages, our approach for constructing the model and validation can be applied to any software quality model.

To improve the practical use of the model, we then employed our model to investigate ways to proactively improve the quality of model transformations. Here, a focus on developer tooling was taken, and many potential directions for tooling based on the validated quality model were identified. One of the tools identified as most useful, a code coverage tool for QVTo, was then developed using an iterative development process with practitioner involvement. During tool design, code coverage criteria were defined specifically for the QVTo language. The tool was evaluated by having a team use it in practice and then conducting a qualitative evaluation of the tool combined with a timing analysis. In addition to now being used by the team cooperating with this research, this tool has also been open sourced and is freely available to the QVTo community. Areas for future research with respect to the coverage tool were also identified. Both the tool development process and evaluation methods can also be applied to future work in software quality tooling.

The coverage tool together with the quality model developed here provide a sound basis for future research in QVTo quality assessment and improvement. Future work includes performing additional validation of the quality model, for example that used by van Amstel (2012), conducting additional interviews, and assessing the appropriateness of the quality evaluation procedures. Furthermore, while the quality properties we have identified have been formulated to express directionality, e.g., “High test coverage” or “Small function size,” we consider as another direction of future work determining thresholds for those metrics (Oliveira et al. 2014) and developing appropriate aggregation techniques, allowing one to lift the quality assessment to larger units (Mordal et al. 2013; Vasilescu et al. 2011). Future work for QVTo quality tooling includes addressing the tool threats to validity and developing the additional tools identified in this research based on the QVTo quality model. Finally, to strengthen the relationship between practice and theory, further investigation should be performed on the relationships between the quality properties and best practices presented here.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

(2013a). Google scholar. http://scholar.google.com/intl/en/scholar/about.html
(2013b). Transformation tool contest. http://www.transformation-tool-contest.eu
(2014a). ASML N.V. http://www.asml.com
(2014b). EclEmma: Control flow analysis for java methods. http://www.eclemma.org/jacoco/trunk/doc/flow.html
(2014c). EclEmma: Java code coverage for Eclipse. http://www.eclemma.org/
(2014d). Eclipse community forum QVT-OML. http://www.eclipse.org/forums/index.php/f/244
(2014e). Eclipse plugin for measuring QVTo test coverage. https://github.com/phoxicle/qvto-coverage
(2014f). Java code coverage: Reasons for huge performance impact. http://comments.gmane.org/gmane.comp.java.jacoco.user/66
(2014g). Karlsruhe institute of technology—QVT. https://sdqweb.ipd.kit.edu/wiki/QVT

(2014h). Licenses. http://choosealicense.com/licenses/
(2014i). Patch for QVTo engine: Adding a visitor decorator class. https://github.com/eclipse/qvto/commit/51028ae23d78e9d2b7832321254487458d8e3da7
(2014j). Patch for QVTo engine: Adding hooks for third-party decorators. https://github.com/eclipse/qvto/commit/8160dd9f29509d7051e4961b36eeaea61fe7a377
(2014k). Patch for QVTo engine: Fixing visitation of imported transformations. https://github.com/eclipse/qvto/commit/d1aa7b9f5ca4c35d36f70031c889b7feec997ed7
Amrani, M., Dingel, J., Lambers, L., Lúcio, L., Salay, R., Selim, G., Syriani, E., & Wimmer, M. (2012). Towards a model transformation intent catalog. In *Proceedings of the first workshop on the analysis of model transformations, ACM* (pp. 3–8).
Barendrecht, P. J. (2010). Modeling transformations using QVT operational mappings. Master's thesis, Technische Universiteit Eindhoven. http://redpanda.nl/BEP_P.J.Barendrecht.pdf. Accessed April 1, 2014.
Barnette, J. J. (2000). 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.
Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., et al. (2001). Manifesto for agile software development. http://www.agilealliance.org/the-alliance/the-agile-manifesto/.
Ciancone, A., Filieri, A., & Mirandola, R. (2010). Mantra: Towards model transformation testing. In *2010 seventh international conference on the quality of information and communications technology (QUATIC), IEEE* (pp. 97–105).
Del Fabro, M. D., & Valduriez, P. (2009). Towards the efficient development of model transformations using model weaving and matching transformations. *Software & Systems Modeling*, *8*(3), 305–324.
Easterbrook, S., Singer, J., Storey, M. A., & Damian, D. (2008). Selecting empirical methods for software engineering research. In F. Shull, J. Singer, & D. I. K. Sjøberg (Eds.), *Guide to advanced empirical software engineering* (pp. 285–311). London: Springer.
Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., & Varró-Gyapay, S. (2005). Termination criteria for model transformation. In M. Cerioli (Ed.), *Fundamental approaches to software engineering* (Vol. 3442, pp. 49–63), Lecture notes in computer science. Berlin: Springer.
Ergin, H., & Syriani, E. (2013). Identification and application of a model transformation design pattern. In *ACM Southeast regional conference*. ACM.
Ferenc, R., Hegedüs, P., & Gyimóthy, T. (2014). Software product quality models. In T. Mens, A. Serebrenik, & A. Cleve (Eds.), *Evolving software systems* (pp. 65–100). Berlin: Springer.
Field, A. P. (2005). Kendall's coefficient of concordance. In B. Everitt, & D. Howe (Eds.), *Encyclopedia of statistics in behavioral science*. Hoboken: Wiley.
Fowler, M. (1999). *Refactoring: improving the design of existing code*. Reading: Addison-Wesley Professional.
France Telecom. (2014). SmartQVT. https://yoxos.eclipsesource.com/yoxos/node/fr.tm.elibel.smartqvt.feature.group
Gerpheide, C. M. (2014). Assessing and improving quality in QVTo model transformations. Master's thesis, Technische Universiteit Eindhoven. http://alexandria.tue.nl/extra1/afstversl/wsk-i/gerpheide2014.pdf
Gerpheide, C. M., Schiffelers, R. R., & Serebrenik, A. (2014a). A bottom–up quality model for QVTo. In *2014 ninth international conference on the quality of information and communications technology (QUATIC). IEEE.*
Gerpheide, C. M., Schiffelers, R. R., & Serebrenik, A. (2014b). QVTo model transformations: Assessing and improving their quality. *ERCIM Special Theme: Software Quality*, *99*, 32–33.
Gniesser, P. (2012). Refactoring support for ATL-based model transformations. Master's thesis, Faculty of Informatics-Vienna University of Technology.
Guana, V., & Stroulia, E. (2014). Chaintracker, a model-transformation trace analysis tool for code-generation environments. In D. Di Ruscio & D. Varŕo (Eds.), *Theory and practice of model transformations* (pp. 146–153). Switzerland: Springer International Publishing.
Guduric, P., Puder, A., & Todtenhofer, R. (2009). A comparison between relational and operational QVT mappings. In *Sixth international conference on information technology: New generations, 2009. ITNG'09* (pp. 266–271). IEEE.
Hall, T., & Fenton, N. (1997). Implementing effective software metrics programs. *IEEE Software*, *14*(2), 55–65.
Hove, S. E., & Anda, B. (2005). Experiences from conducting semi-structured interviews in empirical software engineering research. In *METRICS, IEEE* (pp. 10–23).
ISO/IEC 25000. (2014). *Systems and software engineering—Systems and software quality requirements and evaluation (SQuaRE)—Guide to SQuaRE*. ISO, Geneva, Switzerland.

ISO/IEC 25010. (2011). *Systems and software quality requirements and evaluation (SQuaRE)—System and software quality models*. ISO, Geneva, Switzerland.

Johns, R. (2010). Likert items and scales. Survey Question Bank: Methods Fact Sheet. http://surveynet.ac.uk/sqb/datacollection/likertfactsheet.pdf

Kapová, L., Goldschmidt, T., Becker, S., & Henss, J. (2010). Evaluating maintainability with code metrics for model-to-model transformations. In *Quality of software architectures (QoSA)*, LNCS (Vol. 6093, pp. 151–166). Springer.

Kitchenham, B. (2004). Procedures for performing systematic reviews. Technical Report TR/SE0401, Keele University.

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

Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J., & Linkman, S. (2009). Systematic literature reviews in software engineering—A systematic literature review. *Information and Software Technology*, *51*(1), 7–15.

Kitchenham, B., & Pfleeger, S. L. (1996). Software quality: The elusive target. *IEEE Software*, *13*(1), 12–21.

Kolahdouz-Rahimi, S., Lano, K., Pillay, S., Troya, J., & Van Gorp, P. (2014). Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming*, *85*, 5–40.

Kusel, A., Schönböck, J., Wimmer, M., Retschitzegger, W., Schwinger, W., & Kappel, G. (2013). Reality check for model transformation reuse: The ATL transformation zoo case study. In *2nd workshop on the analysis of model transformations (AMT) @ MODELS'13 1077*.

Lehrig, S. (2012). Assessing the quality of model-to-model transformations based on scenarios. Master's thesis, University of Paderborn, Zukunftsmeile 1.

Lin, Y., Zhang, J., & Gray, J. (2005). A testing framework for model transformations. In S. Beydeda, M. Book, & V. Gruhn (Eds.), *Model-driven software development* (pp. 219–236). Berlin, Heidelberg: Springer.

McQuillan, J. A., & Power, J. F. (2009). White-box coverage criteria for model transformations. In *First international workshop on model transformation with ATL*.

Mens, T., & Van Gorp, P. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, *152*, 125–142.

Mohagheghi, P., & Dehlen, V. (2007). An overview of quality frameworks in model-driven engineering and observations on transformation quality. In *Workshop on quality in modeling* (p. 3).

Moody, D. L. (2003). Measuring the quality of data models: An empirical evaluation of the use of quality metrics in practice. In *European conference on information systems (ECIS)* (pp. 1337–1352).

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

Mordal, K., Anquetil, N., Laval, J., Serebrenik, A., Vasilescu, B., & Ducasse, S. (2013). Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*, *25*(10), 1117–1135.

Nguyen, P. H. (2010). Quality analysis of model transformations. Master's thesis, Technische Universiteit Eindhoven.

Nolte, S. (2010). QVT-operational mappings. QVT-operational mappings: Modellierung mit der query views transformation, Xpert press. ISBN 978-3-540-92292-6. Berlin: Springer, 2010 1.

Oliveira, P., Valente, M. T., & Lima, F. P. (2014). Extracting relative thresholds for source code metrics. In S. Demeyer, D. Binkley, F. Ricca (Eds.), *European conference on software maintenance and reengineering—Working conference on reverse engineering (CSMR-WCRE), IEEE* (pp. 254–263).

OMG. (2011). MOF 2.0 Query/View/Transformation Spec. V1.1.

OMG. (2012). Object constraint language.

Orejas, F., Guerra, E., De Lara, J., & Ehrig, H. (2009). Correctness, completeness and termination of pattern-based model-to-model transformation. In A. Kurz, M. Lenisa, & A. Tarlecki (Eds.), *Algebra and coalgebra in computer science* (pp. 383–397). Berlin, Heidelberg: Springer.

Paige, R. F., & Varró, D. (2012). Lessons learned from building model-driven development tools. *Software & Systems Modeling*, *11*(4), 527–539.

Planas, E., Cabot, J., & Gómez, C. (2011). Two basic correctness properties for ATL transformations: Executability and coverage. In *3rd international workshop on model transformation with ATL*, Zurich, Switzerland.

Rahim, L. A., & Whittle, J. (2015). A survey of approaches for verifying model transformations. *Software & Systems Modeling, 14*, 1003–1028.

Ramamoorthy, C., & Ho, S. F. (1975). Testing large software with automated software evaluation systems. In *ACM SIGPLAN notices, ACM* (Vol. 10, pp. 382–394).

Rentschler, A., Noorshams, Q., Happe, L., & Reussner, R. (2013a). Interactive visual analytics for efficient maintenance of model transformations. In K. Duddy & G. Kappel (Eds.), *Theory and practice of model transformations* (pp. 141–157). Berlin, Heidelberg: Springer.

Rentschler, A., Noorshams, Q., Happe, L., & Reussner, R. (2013b). Interactive visual analytics for efficient maintenance of model transformations. In *International conference on model transformation (ICMT)*. LNCS (Vol. 7909, pp. 141–157). Springer.

Rose, L. M., Herrmannsdoerfer, M., Mazanek, S., Van Gorp, P., Buchwald, S., Horn, T., et al. (2014). Graph and model transformation tools for model migration. *Software & Systems Modeling, 13*, 323–359.

Santiago, I., Vara, J. M., de Castro, V., & Marcos, E. (2013). Measuring the effect of enabling traces generation in atl model transformations. In *Evaluation of novel approaches to software engineering* (pp. 229–240). Springer.

Schiffelers, R. R., Alberts, W., & Voeten, J. P. (2012). Model-based specification, analysis and synthesis of servo controllers for lithoscanners. In *International workshop on multi-paradigm modeling, ACM* (pp. 55–60).

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

Selim, G. M., Cordy, J. R., & Dingel, J. (2012a). Analysis of model transformations. PhD thesis, Technical Report, Queen's University School of Computing.

Selim, G. M., Cordy, J. R., & Dingel, J. (2012b). Model transformation testing: The state of the art. In *Proceedings of the first workshop on the analysis of model transformations, ACM* (pp. 21–26).

Stahl, T., & Voelter, M. (2006). *Model-driven software development*. Chichester, England: Wiley.

Syriani, E., & Gray, J. (2012). Challenges for addressing quality factors in model transformation. In *International conference on software testing (ICST), IEEE* (pp. 929–937).

van Amstel, M. F. (2012). Assessing and improving the quality of model transformations. PhD thesis, Technische Universiteit Eindhoven.

van Amstel, M. F., & van den Brand, M. G. (2011). Model transformation analysis: Staying ahead of the maintenance nightmare. In J. Cabot & E. Visser (Eds.), *Theory and practice of model transformations* (pp. 108–122). Berlin, Heidelberg: Springer-Verlag.

van Amstel, M. F., Bosems, S., Kurtev, I., & Pires, L. F. (2011). Performance in model transformations: Experiments with ATL and QVT. In J. Cabot & E. Visser (Eds.), *Theory and practice of model transformations* (pp. 198–212). Berlin, Heidelberg: Springer-Verlag.

van Amstel, M. F., van den Brand, M. G. J., & Nguyen, P. H. (2010). Metrics for model transformations. In *Proceedings of the ninth Belgian-Netherlands software evolution workshop (BENEVOL 2010)*, Lille, France.

van Amstel, M. F., van den Brand, M. G. J., & Serebrenik, A. (2012). Traceability visualization in model transformations with TraceVis. In Z. Hu & J. de Lara (Eds.), *Theory and practice of model transformations* (pp. 152–159). Berlin, Heidelberg: Springer-Verlag.

van Dongen, M. (2012). Visualization of model transformations in QVTo. Master's thesis, Technische Universiteit Eindhoven.

Vasilescu, B., Serebrenik, A., & van den Brand, M. G. J. (2011). By no means: A study on aggregating software metrics. In *2nd international workshop on emerging trends in software metrics, ACM, WETSoM* (pp. 23–26).

Vidmar, G., & Rode, N. (2007). Visualising concordance. *Computational Statistics*, *22*(4), 499–509.

Vignaga, A. (2009). Metrics for measuring ATL model transformations. Technical Report, Department of Computer Science, Universidad de Chile.

Voelter, M., Kolb, B. (2006). Best practices for model-to-text transformations. In *In Eclipse Summit Europe, modeling symposium* (Vol. 2006, p. 27).

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

Christine M. Gerpheide is a software development engineer at Amazon Web Services. She completed her Master of Science in Computer Science and Engineering cum laude and with Honors from Eindhoven University of Technology. In her career Christine has worked on a wide range of software, including service-oriented architectures, model-driven engineering, and mobile development

Dr.ir. Ramon R. H. Schiffelers is a software architect at ASML N.V. and an assistant professor at Eindhoven University of Technology within the research group for model-driven software engineering.

Dr. Alexander Serebrenik is an associate professor of Model-Driven Software Engineering at Eindhoven University of Technology. He has obtained his Ph.D. in Computer Science from Katholieke Universiteit Leuven, Belgium (2003) and M.Sc. in Computer Science from the Hebrew University, Jerusalem, Israel. Dr. Serebrenik's areas of expertise include software evolution, maintainability and reverse engineering, program analysis and transformation, process modeling and verification.