The Journal of Systems & Software 169 (2020) 110684

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A comparison of quality flaws and technical debt in model transformation specifications

Shekoufeh Kolahdouz-Rahimi [a,*], Kevin Lano [b], Mohammadreza Sharbaf [a], Meysam Karimi [a], Hessa Alfraihi [c]

[a] MDSE Research Group, Dept. of Software Engineering, University of Isfahan, Iran
[b] Dept. of Informatics, King's College London, London, UK
[c] Dept. of Information Systems, Princess Nourah bint Abdulrahman University, Saudi Arabia

ARTICLE INFO

Article history:
Received 20 January 2020
Received in revised form 29 May 2020
Accepted 1 June 2020
Available online 8 June 2020

Keywords:
Model transformations
Technical debt
Software quality

ABSTRACT

The quality of model transformations (MT) has high impact on model-driven engineering (MDE) software development approaches, because of the central role played by transformations in MDE for refining, migrating, refactoring and other operations on models.

For programming languages, a popular paradigm for code quality is the concept of *technical debt* (TD), which uses the analogy that quality flaws in code are a debt burden carried by the software, which must either be 'redeemed' by expending specific effort to remove its flaws, or be tolerated, with ongoing additional costs to maintenance due to the flaws.

Whilst the analysis and management of quality flaws and TD in programming languages has been investigated in depth over several years, less research on the topic has been carried out for model transformations. In this paper we investigate the characteristics of quality flaws and technical debt in model transformation languages, based upon systematic analysis of over 100 transformation cases in four leading MT languages.

Based on quality flaw indicators for TD, we identify significant differences in the level and kinds of technical debt in different MT languages, and we propose ways in which TD in MT can be reduced and managed.

1. Introduction

This paper will investigate the issue of quality flaws and *technical debt* (TD) (Cunningham, 1992; Marinescu, 2012) in model transformation specifications (MT). The original definition of TD in Cunningham (1992) referred to the use of non-optimal coding practices used to accelerate development, and the consequences of these practices: "shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite [...] The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt".

An example of such 'quick and dirty' practices is the use of copy–paste to define a new operation based on an existing operation, instead of generalising the existing operation.

Thus technical debt refers to the *additional costs incurred for the maintenance of a software artefact over its lifetime, caused by the use of non-optimal practices in its production*. The *principal* cost of TD is incurred when refactoring or other remedial action is used to remove the TD from the software, whilst the *interest* is paid in the additional cost due to the TD each time the software is maintained. Depending on the projected lifetime of the artefact, there may be lower total costs in tolerating the interest payments as opposed to paying off the debt.

There are a range of perspectives on what constitutes technical debt (Tom et al., 2012) and how it may be estimated (Zazworka et al., 2014). The 'non-optimal practices' in software production can encompass failures to use appropriate architectures or design patterns in the software artefact, functional or testing incompleteness, inappropriate and inadequate documentation, insufficient consultation with stakeholders, and others (Zazworka et al., 2014).

The concept of TD was initially applied to code artefacts, but can also be extended to analysis and design models (Arendt and Taentzer, 2010). In Model Driven Engineering (MDE) (Brambilla et al., 2012; Lano, 2016), model transformations play a central role, supporting MDE processes such as the synthesis of

* Corresponding author.
E-mail addresses: sh.rahimi@eng.ui.ac.ir (S. Kolahdouz-Rahimi), kevin.lano@kcl.ac.uk (K. Lano), m.sharbaf@eng.ui.ac.ir (M. Sharbaf), meysam.karimi@eng.ui.ac.ir (M. Karimi), hessa.alfraihi@kcl.ac.uk (H. Alfraihi).

https://doi.org/10.1016/j.jss.2020.110684
0164-1212/

software and documentation from models, and model synchronisation and comparison. Thus the quality and maintainability of MT specifications are relevant for the successful use of MDE.

The goal of our research is to *quantify and characterise the nature of technical debt in model transformations*. We use the goal-question-metric (GQM) approach of Basili (1992) to decompose this goal into specific questions and metrics. The goal results in the following research questions:

RQ1: What is the extent of technical debt in MT cases?

RQ2: What are the most frequent technical debt issues in MT cases?

RQ3: How does the level and character of TD vary between MT languages and between different MT application categories?

RQ4: What are the differences between the extent and character of TD in MT languages and in traditional programming languages?

Formally, the TD of version v_s of a software product released at time T can be defined as the additional effort Δw needed to maintain the product until the end of its lifetime (version v_e, which is decommissioned at time $T + \Delta t$) caused by non-optimal practices in the development history of v_s.

It is difficult to measure Δw directly. Developers would need to precisely track effort spent on maintenance work on the product, identifying where additional costs were incurred due to deficiencies in the product code/documentation of the v_s version.

Instead of directly measuring technical debt, we will use the frequency of occurrence of various quality flaws in the studied transformations as indicators of the level of technical debt that they carry. These indicators include occurrences of duplicated specification text, excessive dependencies or overlarge/over-complex rules or operations. We choose this approach because the analysis can be automated based purely on the transformation text (and on any metamodels that it depends upon), whilst there may be insufficient information to take account of factors such as functional or requirements incompleteness. The approach is commonly used in TD analysis of code, for example (Digkas et al., 2017; He et al., 2016; Letouzey, 2016; Marinescu, 2012). Such approaches therefore analyse the TD due to code quality flaws (rather than due to other factors). In Zazworka et al. (2014) this approach is compared (for Java code) to other TD identification techniques. They found that some measures of coupling, size and modularity flaws were good predictors of which classes were more defect-prone and change-prone. Kosti et al. (2017) also provide evidence that measures of software quality flaws are good predictors of TD. We will evaluate the appropriateness of our indicators for TD in MT by analysing whether the changes made to some sample transformations over time have been focussed on the parts of the transformations identified as having high TD according to their quality flaws (Section 7).

We will therefore consider the research questions **RQ1** to **RQ4** with respect to TD as inferred from quality flaws. The questions imply that a substantial sample of transformations must be surveyed, for a range of transformation languages and categories. We will use published and machine-readable transformation cases, and public repositories of transformations.[1] Only cases where the complete code of the transformations is available will be considered. We survey the ATL (Eclipse, 2019) and ETL (Kolovos et al., 2008) transformation languages because these are widely-used by practitioners (Batot et al., 2016; Burgueno et al., 2019). We also consider QVT-R (OMG, 2016) and UML-RSDS (Lano, 2016), which are MT languages with distinctive features (bidirectional execution facilities in the case of QVT-R, and no rule–rule dependencies in UML-RSDS) whose impact on TD levels is of interest. The authors have extensive experience of using UML-RSDS and ETL (6+ years) and some experience of ATL and QVT-R (3+ years).

Additionally, we consider to what extent strategies for managing and reducing technical debt in programming languages can be adapted to apply to MT (Appendix B).

Section 2 describes indicators for technical debt in model transformations, based on a general quality model for MT languages. Section 3 describes our experimental procedure. Section 4 gives an overview of the four specific MT languages considered in the paper. Section 5 presents the analysis results for each language. Section 6 discusses the results and considers how these address the research questions. Section 7 evaluates how MT technical debt changes as transformations evolve. Section 8 considers threats to the validity of our findings. Section 9 surveys related work, and Section 10 describes potential future research directions. Section 11 gives conclusions.

Appendix A provides detailed analysis data, Appendix B considers how technical debt can be managed and reduced for model transformations and Appendix C describes tool support for TD measurement in MT.

2. Indicators for technical debt

Following on from the research questions, we identify key quality requirements for model transformations, which are considered to impact on technical debt in MT. Failures to achieve these quality requirements will be quantified by concrete measures for specific quality flaws. Counts of the number of failure occurrences will be used as indicators of the level of technical debt. In this paper we will only consider quality requirements which are relevant to all of the studied MT languages. Additional specific requirements could be added for the analysis of individual languages.

2.1. Model transformation quality models

A *quality model* for software defines quality requirements for software artefacts and relates these to quality characteristics such as *Reliability*, *Performance efficiency*, *Usability*, *Security*, *Maintainability*, *Portability*, *Compatibility* and *Functional suitability* (IEC/ISO, 2011). Measures are defined for each quality requirement, with thresholds to define acceptable ranges of values of the measures. Violations of the thresholds are counted as quality flaws.

We will follow the SQALE method approach (Letouzey, 2016), which adopts characteristics *Testability*, *Reliability*, *Changeability*, *Efficiency*, *Usability*, *Security*, *Maintainability*, *Portability* and *Reuseability*. Each quality requirement is associated with the quality characteristics that it affects. Violations of quality requirements by a software artefact – i.e., quality flaws – are counted as indicators of the presence of technical debt in the artefact.

Based on the existing literature on MT quality (van Amstel and van den Brand, 2011c; Bonet et al., 2018; Gerpheide et al., 2016; Kapova et al., 2010; Kolahdouz-Rahimi et al., 2014; Wimmer et al., 2012), and on our own experience of developing and maintaining MT specifications, we identified the following categories of quality requirements for MT specifications: size; semantic complexity (of expressions, rules and operations); complexity of dependencies and degree of coupling between transformation elements (rules or operations); redundancy (eg., no duplicated or unused code); performance (use of efficient coding practices);

[1] Such as the ATL zoo atwww.eclipse.org/atl/atlTransformations.

Table 1
Model transformation quality measures.

Category	*Quality measures*
Size	Transformation module LOC and c measure Kolahdouz-Rahimi et al. (2014) Number of rules, *nrules* Number of operations, *nops* Rule, operation size in LOC and c
Semantic Complexity	*#parameters*, *#local variables* Number of alternative rule execution orders Cyclomatic complexity McCabe and Watson (1994)
Dependencies/ Coupling	Operation/rule fan-out and fan-in Total number of calling dependencies Total number of elements in calling cycles
Redundancy	Number of clones of size 10 tokens or more

style (such as appropriate use of design patterns and language features).

A large number of MT metrics have been defined in the literature (van Amstel and van den Brand, 2011c; van Amstel and van den Brand, 2011b; Gerpheide et al., 2016; Kapova et al., 2010), however many of these are language-specific. We selected a small number of key measures which are suitable for analysing quality flaws, and which are independent of MT language. Table 1 lists the measures used.

In this paper we do not consider performance or style quality requirements, or measures for these requirements, but these are important topics for future work, which we discuss in Section 10.

Table 2 summarises the MT quality model adopted in terms of quality requirements, the interpretation of these requirements as constraints on the measures of Table 1, and the affected quality characteristics.

Quality flaws can also be regarded as 'code smells' in the transformations, in the sense of Fowler and Beck (2019), He et al. (2016), Lacerda et al. (2020). Thus we name the TD indicators corresponding to quality flaws based on well-known terminology for code smells (Table 3).

Quality measure values outside the thresholds identify violations of the quality requirements, and have impact on the quality characteristics. All of the size-related flaws increase the cost of maintenance, by increasing the extent of the specification which needs to be analysed and understood to carry out a maintenance action. For the same reason they impact on the effort of testing and of making specification changes.

For semantic complexity, EPL flaws increase the testing effort for the artefact, as tests must consider values for each variable. Understandability is potentially reduced, increasing the effort needed to carry out changes or other maintenance actions in the artefact.

Similarly to EPL, excessive CC increases the testing effort because each execution path needs to be tested, which means each combination of basic conditions needs to be considered. The effort for understandability and hence the effort needed to make changes are likewise increased.

UEX may reduce reliability because rules may execute in unintended orders, resulting in incorrect behaviour. It also increases the testing effort as all possible execution orders need to be tested.

For the dependencies/coupling category, all of the coupling flaws affect changeability, by increasing the number of elements which are affected by a change in a given element. Similarly, they also increase the effort involved in testing and maintaining the transformation. Elements which depend on several other elements are difficult to reuse, especially if there are mutual dependencies involving the element.

For redundancy, code clones result in potential duplication of testing and maintenance effort, and the risk of changes being made to one clone but not to another, thus affecting functional correctness.

The relative impact of different quality flaws or types of technical debt is a subjective issue, and difficult to quantify. We will therefore only provide counts of the indicators and not weight

Table 2
Model transformation quality model.

Category	*Quality requirement*	*Interpretation*	*Quality characteristics* Letouzey (2016)
Size	Transformation modules τ should not be of excessive size	$c(\tau) \leq 1000$ or $LOC(\tau) \leq 500$	Maintainability, Changeability
	Transformations should not have excessive numbers of rules or operations	$nrules \leq 10$ and $nops \leq 10$	Maintainability, Testability, Changeability
	Rules and operations should not be of excessive size.	$c(rule) \leq 100$, $c(op) \leq 100$ or $LOC(rule) \leq 50$, $LOC(op) \leq 50$	Maintainability
Semantic Complexity	Rules/operations should not have more than 10 parameters including local variables	*#parameters + #local variables* ≤ 10	Testability, Changeability, Maintainability
	There should not be high numbers of alternative execution orders	Alternative orders ≤ 10	Reliability, Testability
	Rules and operations should not have excessive cyclomatic complexity	Cyclomatic complexity ≤ 10 for any rule/operation	Testability, Changeability
Dependencies/ Coupling	Operations/rules should not call more than 5 different operations	*fan-out* ≤ 5	Testability, Changeability, Maintainability, Reusability
	Operations should not be called by more than 5 different operations/rules	*fan-in* ≤ 5	Changeability
	In a transformation with r rules and n operations, there should be no more than $r + n$ calling dependencies	Calling dependencies $\leq r + n$	Testability, Changeability, Maintainability
	There should be no cycles in the graph of calling dependencies	Number of elements in call cycles = 0	Testability, Changeability, Maintainability, Reusability
Redundancy	There should be no clones of size 10 tokens or more	*#clones* $= 0$	Testability, Maintainability, Reliability

Table 3
Technical debt indicators and thresholds.

Indicator	*Description*	*Threshold for flaw/code smell*
ETS	Excessive transformation size	$c(\tau) > 1000$, or length > 500 LOC
ENR	Excessive number of rules	$nrules > 10$
ENO	Excessive number of helpers/operations	$nops > 10$
ERS	Excessive rule size	$c(r) > 100$ or length greater than 50 LOC
EHS	Excessive helper size	$c(h) > 100$ or length > 50 LOC
EPL	Excessive parameter list	> 10 parameters including auxiliary rule/operation variables
UEX	Undefined execution orders	> 10 alternative rule orderings
CC	High cyclomatic complexity	≥ 10 basic control flow conditions in an element
EFO	Excessive fan-out	> 5 different rules/operations called from one rule/operation.
EFI	Excessive fan-in	> 5 different rules/operations call one rule/operation
CBR	Coupling between rules	
CBR_1	Excessive coupling	#client–supplier relations $>$ $nrules + nops$
CBR_2	Cyclic dependencies	Number of elements in cyclic dependencies > 0
DC	Duplicate code/expressions	Any duplicated x with $t(x) > 10$

Table 4
OCL expression complexity measures (adapted from Kolahdouz-Rahimi et al. (2014)).

Expression e	*Complexity c(e)*	*Token count t(e)*
Numeric, boolean or String value	0	1
Identifier *iden* Basic expression *obj.f*	1 $c(obj) + c(f) + 1$	1 $t(obj) + t(f) + 1$
Operation call $e(p1, \ldots, pn)$	$c(e) + 1 + \Sigma_i c(pi)$	$t(e) + n + 1 + \Sigma_i t(pi)$
Unary expression *op e* $e \rightarrow op()$	$c(e) + 1$	$t(e) + 1$ $t(e) + 4$
Binary expression *e1 op e2* $e1 \rightarrow op(e2)$	$c(e1) + c(e2) + 1$	$t(e1) + t(e2) + 1$ $t(e1) + t(e2) + 4$
Ternary expression $op(e1, e2, e3)$ *if e1 then e2 else e3 endif*	$c(e1) + c(e2) + c(e3) + 1$	$t(e1) + t(e2) + t(e3) + 5$ $t(e1) + t(e2) + t(e3) + 4$
let $v : T = e1$ *in e2*	$c(T) + c(e1) + c(e2) + 4$	$t(T) + t(e1) + t(e2) + 5$
Set$\{e1, \ldots, en\}$ *Sequence*$\{e1, \ldots, en\}$	$1 + \Sigma_i c(ei)$	$2 + n + \Sigma_i t(ei)$

these differently. In Appendix B we consider how the effort for removing different forms of TD can be quantified, based on the SQALE model (Letouzey and Ilkiewicz, 2012; Letouzey, 2016).

2.2. Choice of measures and thresholds

The size of software artefacts is usually measured in terms of lines of code (LOC). However this is not suitable for comparing artefacts in different software languages because it depends on the formatting/structure of each language. Instead we adopt a language-independent measure $c(\tau)$ of the semantic content of a model transformation specification τ, based on the complexity of expressions/statements in the transformation. In Kolahdouz-Rahimi et al. (2014) we found that complexity correlated more strongly than LOC with measures of development effort and usability. Each of ATL, ETL, QVT-R and UML-RSDS have similar expression languages based on OCL, and ATL, ETL and UML-RSDS have similar pseudocode-style statement languages. Therefore $c(\tau)$ can be defined consistently for all these languages. Table 4 summarises the semantic complexity measure $c(e)$ for some OCL expressions e. $c(e)$ can be considered a count of the number of basic semantic elements in a specification (identifiers plus composite expressions). For example, $c(role \rightarrow select(x|x.att = value) \rightarrow size())$ is 7, for identifiers *role* and *att*, treating $s \rightarrow select(x|e)$ as a binary operator with arguments s and e. c is closely related to Halstead's *program length* measure (Halstead, 1977), the total number of operators + operands in a programme. However, we do not count occurrences of literal constants. A similar measure c can be given to statements (Table 5 shows the values for UML-RSDS syntax, similar definitions can be given for the ATL and ETL statement syntaxes).

Table 5
Statement complexity measures (adapted from Kolahdouz-Rahimi et al. (2014)).

Statement s	*Complexity c(s)*	*Token count*
`return` e	$c(e) + 1$	$t(e) + 1$
$v := e$	$c(v) + c(e) + 1$	$t(v) + t(e) + 1$
$s1; s2$	$c(s1) + c(s2) + 1$	$t(s1) + t(s2) + 1$
Operation call $e(p1, \ldots, pn)$	$c(e) + 1 + \Sigma_i c(pi)$	$t(e) + n + 1 + \Sigma_i t(pi)$
`if` e `then` $s1$ `else` $s2$	$c(e) + c(s1) + c(s2) + 1$	$t(e) + t(s1) + t(s2) + 3$
`for` $v : e$ `do` s	$c(e) + c(s) + 1$	$t(e) + t(v) + t(s) + 3$
`while` e `do` s	$c(e) + c(s) + 1$	$t(e) + t(s) + 2$
`break`	1	1
`continue`	1	1
`var` $v : T$	$c(T) + 3$	$t(T) + 3$

Using these measures, $c(r)$ for a transformation rule r is taken as the sum of the c measures of its parts (such as the *from*, *to* and *do* clauses in ATL), likewise for operation definitions. The semantic complexity $c(\tau)$ of a transformation is taken as the sum of the complexities of its rules and operations.

In addition to complexity (c), we also consider the LOC measure of size because this is widely used for TD estimation. We will

evaluate flaw density both wrt LOC and complexity. We adopt 50 LOC per rule/operation and 500 LOC per transformation as size thresholds, for size measured by LOC. These thresholds apply to ATL, ETL and QVT-R. For UML-RSDS we adopt limits based on c (100 and 1000 respectively). These limits are based on our experience with maintenance of UML-RSDS transformations. A commonly-used threshold for excessive module or class length in programming languages is 1000 LOC (Letouzey, 2016), and for excessive method length is 100 LOC. Since transformation specifications are typically more concise and semantically dense than programming code, lower LOC thresholds are appropriate for MT modules and rules.

The threshold 10 for the EPL indicator is adopted from the PMD code analyser (https://pmd.github.io), as is the threshold 10 for ENR, ENO and CC. An ENO threshold of 11 is used in Zazworka et al. (2014) based on empirical analysis. A CC limit of 10 was originally suggested by McCabe (1976). This CC limit is also used for Java in the SQALE method examples of Letouzey (2016), although with a lower EPL threshold of 5. EPL, ENR, ENO, CC affect maintainability, changeability and testability in a similar way in both programme code and in transformations, thus it is reasonable to adopt similar thresholds. Regarding *UEX*, whilst under-determined execution order of rules can be useful for efficiency and abstraction of transformation execution, excessive use of this feature can hinder comprehension and testability of the transformation, by increasing the number of possible execution paths. Therefore we set a threshold of 10 for *UEX*.

Coupling also impacts software quality in similar ways in programming and MT languages (especially affecting understandability, changeability and testability) but MT languages provide additional forms of coupling (such as implicit invocation) and dependency mechanisms compared to programming languages. In this paper we weight all of the different forms of coupling equally, but there may be an argument that implicit forms of coupling should be weighted more highly than explicit (because it requires additional effort to understand what are the actual components invoked by an implicit call). Instead of adopting a fixed threshold for coupling (such as 20 in PMD or 6 in SQALE for Java), we compare the number CBR_1 of calling dependencies between rules/operations with the total number of rules and operations. If the call graph structure is complex, with more edges than nodes, CBR_1 is considered to be over the threshold. The number CBR_2 of rules/operations occurring in mutually-dependent groups is also separately counted.

We define a token count measure $t(e)$ in Tables 4 and 5, which is used for clone detection. Number of tokens is used because in this case value expressions should be counted as contributing to the clone. The lower limit for clone size is set to avoid trivial clones. It could be reduced, at the cost of increased processing time. In Struber et al. (2017) clones of any size are considered. In He et al. (2016), a threshold of 50 tokens is used for code clone detection. We experimented with using 25 tokens as the threshold, but this led to many significant clones being ignored, and we adopted 10 tokens for our analysis. Only identical clones which form a valid expression, statement or other syntactic unit are counted.

We discuss the validity of the selected thresholds, and the effect of changing them, in Section 8.

2.3. Transformation categories and styles

There are factors of intrinsic complexity in certain categories of transformation problems, which consequently affect the size and complexity of transformations that solve these problems. In general, the higher the structural and/or semantic distance between the source and target metamodels, the more complex is the necessary transformation logic.

We give an indication of the intrinsic complexity of each transformation case by classifying its category based on the transformation intent definitions of Lucio et al. (2016):

(1) **Migration** — mapping models from one metamodel to another at the same level of abstraction. This can be subdivided into:
   (1) *Copy* transformations where source and target metamodels are the same or isomorphic
   (2) *Evolution* cases where one metamodel is an evolved version of the other
   (3) *Heterogeneous migration*, where the metamodels are unrelated by evolution.

(2) **Analysis** — extracting information from a model as a view or other analysis result.
(3) **Abstraction** — the inverse of refinement.
(4) **Refinement** — mapping from a higher abstraction level model to a lower-level model.
(5) **Code generation** — mapping from a model to text or to executable code.
(6) **Refactoring** — Update-in-place transformations which restructure a model, retaining its conformance to the same or a closely-related metamodel.
(7) **Semantic mapping** — maps a model m in one language to a formal representation in a language with a formal semantics, to support semantic analysis of m.
(8) **Bidirectional (Bx)** — transformations which can be applied in either source-to-target or target-to-source directions, supporting model synchronisation and change-propagation.

In general, we would expect copy transformations to be simpler than evolution cases, evolution cases to be simpler than heterogeneous migration cases, etc.

We also classify the style of a transformation:

- L: Declarative and logical (mainly based on rule specifications, with no imperative coding)
- F: Declarative and functional (mainly based on functions/operations)
- H: Hybrid (involving some imperative coding elements but less than 50 LOC in any element)
- I: Imperative (involving at least 50 LOC of imperative code in some element).

2.4. Scope of analysis

We do not consider problematic issues in the use of OCL (Correa and Werner, 2007; Cabot and Teniente, 2007) — OCL 'smells' such as the use of chained *implies*, 'magic literals', chained *forAll* quantifiers, long chained navigations in expressions, and other constructions which impair the comprehensibility and efficiency of the specification. To the issues of Correa and Werner (2007) we would add problems such as the use of general *iterate* expressions, or explicit use of the *invalid* value (which affect testability). OCL flaws are a semantic complexity factor for which specific metrics could be devised. We count *let* expression variables and *iterate* accumulator variables in our EPL measure. Flaws may also exist in metamodels, such as concrete superclasses, excessive depth of inheritance hierarchies, excessive numbers of subclasses, or multiple inheritance (Bettini et al., 2019; Strittmatter et al., 2016). Since transformations depend on their metamodels, it may also be expected that metamodel flaws could lead to flawed transformations, or to an increase in their technical debt. For example, multiple inheritance and concrete superclasses could lead to increased complexity in rule conditions and actions. Duplicate features in sibling classes could lead to duplicate code in the

Table 6
Selection of cases.

Language	*Surveyed*	*Selected*	*Industrial*
ATL	104	25	1
ETL	288	31	6
QVT-R	64	27	1
UML-RSDS	47	32	3

transformation. This topic is another important area for future investigation. Further quality-flaw based indicators for technical debt, such as the presence of unused rules or operations (Bonet et al., 2018), could also be used. Likewise for the presence of self-declared technical debt (Maldonado et al., 2017; Wehaibi et al., 2016). However we found that instances of such flaws were unusual in the surveyed cases.

3. Study procedure

We used the following procedure to answer the research questions:

- We identified repositories and other sources of complete MT cases for each of the four languages under study.
- We selected cases for each language, taking account of the need to have similar size distributions in the samples for different languages, and the need for a wide range of styles and sizes to support comparative analysis. In particular, we ensured that there were at least two cases in each language for each main category of transformation (refinement, refactoring and migration).
- We applied the metrics on the source text of the cases. The measures were computed by the second author using the Agile UML toolset (https://projects.eclipse.org/projects/modeling.agileuml) on all cases in order to ensure consistency.
- For LOC, all lines of code are included, including comment and white space lines.
- Summary and comparative statistics were computed using Excel and scipy.

3.1. Selection of cases

We collected cases from all available sources, and then filtered these to select comparable datasets of representative transformation cases for the languages.

Our selection criteria were: (i) the complete code of the cases must be available in text form for analysis, including the definitions of their metamodels; (ii) duplicated cases were excluded; (iii) undergraduate-authored solutions and 'toy' examples of less than 25 *c* were excluded; (iv) automatically-synthesised transformations were excluded.

Table 6 summarises the surveyed and selected cases, including cases used for comparison on common examples and for analysis of evolution. We also indicate how many of the cases were industrial: originated by a non-academic organisation or produced in order to meet the needs of such an organisation.

The particular repositories used as sources of cases were:

- ATL: the ATL zoo at www.eclipse.org/atl/ atlTransformations. This contains 103 cases, although only 92 cases had available source code. We selected a subset of 24 cases (26% of the available examples) which were representative in terms of transformation size (LOC) and transformation category.
  We also included a further ATL case to provide a common case for all the languages: class diagram refactoring (*cdrat*) from Kolahdouz-Rahimi et al. (2014).
- ETL: the Eclipse ETL repository git.eclipse.org/c/epsilon/ org.eclipse.epsilon.git/plain/ examples, and the Epsilonlabs repository github.com/epsilonlabs. We used all available cases (13) published on these repositories, which are maintained by the Epsilon developers.
  We also surveyed the *York* (23 cases, which included the above examples) and *Github* datasets (72 transformations) from github.com/ phillipus85/ETLMetrics/. We excluded student and academic cases from the *Uniandes* cases (191 transformations) of this dataset. Unfortunately, many of the cases in the *Github* repository omit metamodels, which prevents their analysis for quality flaws using our tools. Hence only cases for which metamodels are available were selected.
  We also developed two additional cases in ETL (CDO (Alfraihi and Lano, 2018) and class diagram refactoring) to provide common case studies between the languages.
- QVT-R: Some small tutorial examples (29 examples) are available at the Medini QVT site projects.ikv.de/qvt, however our main source was the QVT-D project repository of examples (13) originating mainly from ModelMorf: git.eclipse.org/c/mmt/org.eclipse.qvtd.git. We used all the examples from this repository. There are very few medium/large-sized QVT-R cases available (a search on Github uncovered only 6 cases over 200 LOC). We have instead selected examples (2) from the QVT-R standard, and (17) from published papers (Goldschmidt and Wachsmuth, 2011; Macedo and Cunha, 2016; Westfechtel, 2018a,b; Westfechtel and Buchmann, 2019). We also developed the CDO case and extended the class diagram refactoring case (Kolahdouz-Rahimi et al., 2014) to provide common cases.
- UML-RSDS: we used cases from the zoo at nms.kcl.ac.uk/kevin.lano/zoo (47 cases). Preferentially, we selected cases which had been developed for industrial collaborators (eg., *CDO*), intended for widespread use (the C and Python code generators) or which had undergone external evaluation (the TTC cases: GMF migration, Activity migration, f2p/p2f, TTC14Live, cra, movies and PetriNet to SM).

25 cases were selected for the main analysis for each language. Fig. 1 shows the number of selected cases in each *c* size range for ATL, ETL and UML-RSDS (excluding the two largest UML-RSDS cases and the largest QVT-R case). These sets of cases for the three languages therefore have similar size profiles. All cases are available online at Kolahdouz-Rahimi et al. (2020b).

Transformation repositories usually only provide a single version of each transformation, or alternative contemporaneous examples. Therefore there is no commit history to show the evolution of the transformations. Thus it is difficult to track changes in flaw levels over time, or to identify where changes have been made in transformation evolution steps. We examined 9 cases in ATL, ETL, QVT-R and UML-RSDS where evolution steps were documented (Section 7).

4. Model transformation languages

Table 7 details some of the correspondences of language elements between ATL, ETL, QVT-R and UML-RSDS. This shows that the languages have many commonalities in terms of their essential concepts (rules, operations, rule sources and targets, etc.), but that these are expressed using significantly different language mechanisms and syntactic forms.

ATL is the most widely-used MT language, according to Burgueno et al. (2019). It provides a relatively simple syntax for specifying transformations, and supports both declarative and imperative constructs. Transformations may be written in a variety of styles, however the conventional style is to use declarative matched rules that invoke helper functions or called/lazy rules to

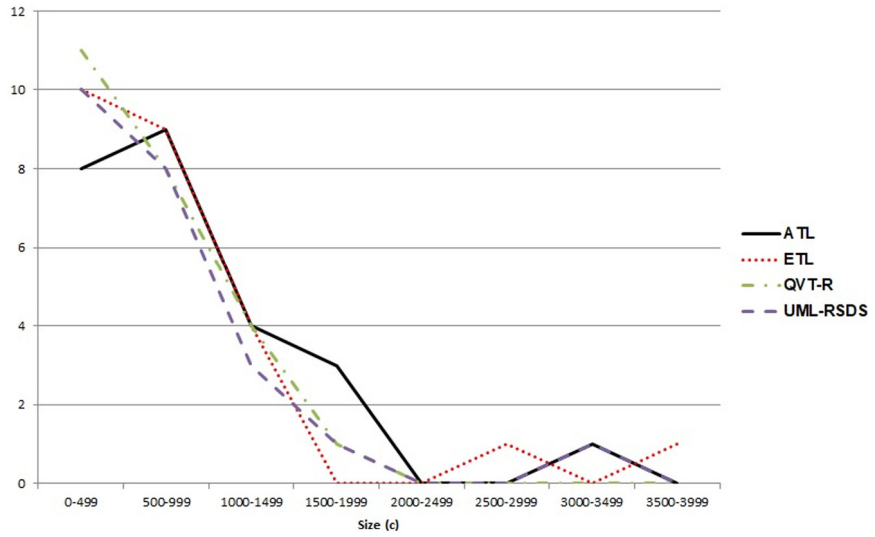

Fig. 1. Distribution of case sizes in ATL, ETL, QVT-R and UML-RSDS.

Table 7
Correspondence of MT languages.

Aspect	*ATL*	*ETL*	*QVT-R*	*UML-RSDS*
Transformation	Transformation module	ETL module	Relational transformation	UML use case
Operation	Helper with entity context	Operation with entity context		Operation of entity
	Helper attribute	Cached operation		Cached operation
	Helper with no context	Variable of transformation	Query function	Operation/attribute of use case
Rule	Called rule	Concrete rule	Relation called in `where` clause	Update operation of use case
	Matched rule (non-lazy)	Non-lazy rule	Top relation	Use case postcondition
	Matched rule (lazy)	Lazy rule	Non-top relation	Operation of use case
Object Resolution	*resolveTemp*(), implicit target element lookup	*equivalent*() ::= implicit rule invocation	`when` $R(s, t)$ clause: Explicit element lookup	$T[key]$, explicit lookup by primary key
Inheritance	Rule inheritance	Rule inheritance	Relation overriding	Operation overriding
Rule Variables	`using` variable	Local rule variable	Local rule variable	Let variable
	In Pattern	Rule source	Non-target domain	Postcondition read frame
	Out Pattern	Rule target	`enforce` target domain	Postcondition write frame
Rule Code	`do` clause	Rule body	`where` clause	Operation activity

carry out parts of their functionality. The standard mode of ATL has a restricted execution semantics (source models are read-only and target models are write-only), which simplifies transformation analysis but restricts the scope of application of ATL. ATL is well-suited to migration and refinement/abstraction tasks, operating on separate source and target models. It is less well-suited for refactoring or other update-in-place tasks. It cannot express bidirectional transformations (bx) except as two separate unidirectional transformations. In addition to its standard mode, ATL also has a *refining* mode, which uses a restricted update-in-place processing model.

A particular issue with ATL specification is the use of *resolveTemp*(e, var) expressions in rules to look up target model elements produced by another rule, during transformation processing. The expression looks up the target element identified by the output variable var of a rule which has mapped e. This reference from one rule to another is considered a semantic complexity factor in van Amstel and van den Brand (2011c) because it introduces a syntactic and semantic dependency of the rule calling *resolveTemp* upon the rule(s) identified by the call. We include the rule-to-rule dependencies induced by *resolveTemp* in the *CBR* measures. Implicit data dependencies between ATL rules are not included in CBR (a rule R is considered to data-depend on rule P if R reads data written by P) in our current analysis. Likewise for the other MT languages we only count calling dependencies and explicit data dependencies as element

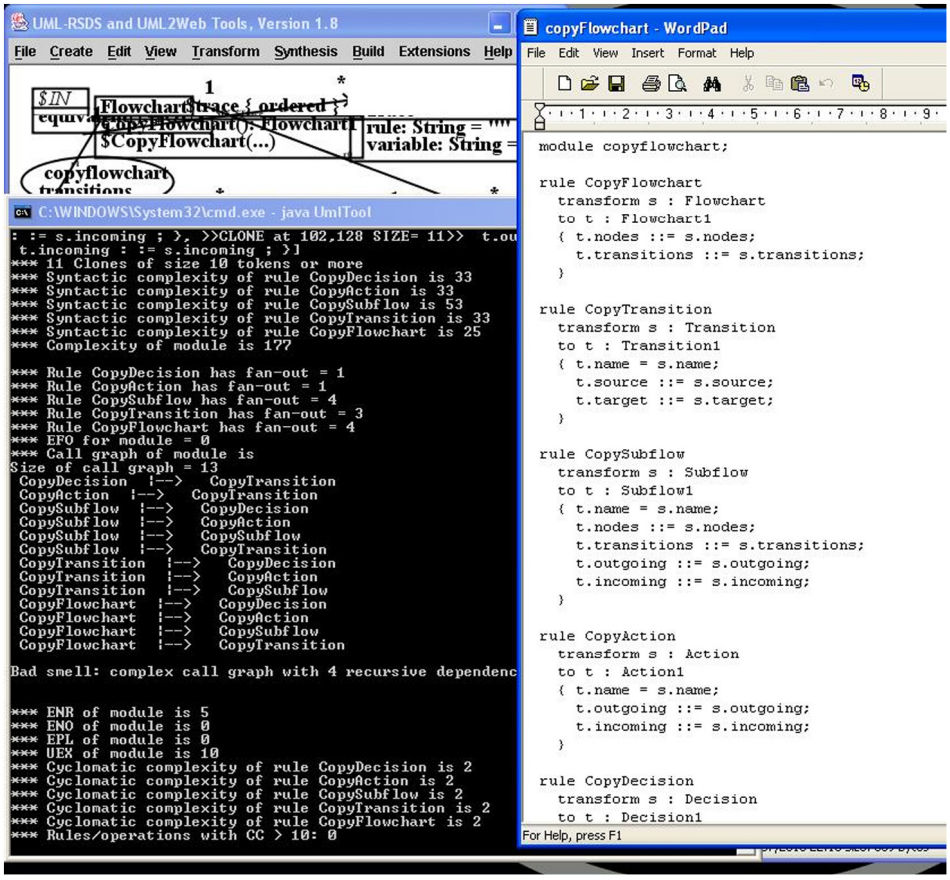

Fig. 2. Example of technical debt analysis (1).

couplings, and do not count implicit data dependencies. For *EPL* we include the `from` and `to` variables of an ATL rule in addition to any rule parameters (for called rules) and local rule variables defined in the `using` clause of a rule.

ETL is the main Epsilon transformation language, and this family of languages is the 3rd-most widely-used for model transformations, according to Burgueno et al. (2019). ETL has a similar rule and transformation structure to ATL, but with a more general processing model and more complex semantics. Target models can be both read and written, so that general refactoring tasks can be handled. Imperative, functional and declarative styles can be used and combined. In contrast to the other MT languages, ETL has implicit invocation of rules by rules or operations: the *equivalent*/*equivalents* expressions can implicitly invoke any concrete rules (lazy or non-lazy) necessary to transform a source element to corresponding target elements. In calculating the call graph and *CBR* measures, such implicit calls must be taken into account. An expression $e.equivalent()$ may lead to the invocation of any concrete lazy or non-lazy rule which has an input variable $v : T$ with T containing the actual value of e *at runtime* (Kolovos et al., 2008). Thus the rule or operation containing the expression can depend upon several concrete rules in the transformation (including itself), potentially leading to large values for fan-in, fan-out and call graph size. The abbreviated form $v ::= e$ of $v = e.equivalent()$ is considered in the same manner. We count `pre` and `post` blocks as rules, with a fixed ordering relation relative to the other rules. Fig. 2 illustrates the issue of rule dependencies in ETL: a small transformation with 5 rules nonetheless has 13 calling dependencies between rules, including 4 mutually dependent rules. These cyclic dependencies are an indicator of a failure to use the Map Objects before Links design pattern (Lano and Kolahdouz-Rahimi, 2014). Duplicate code also arises in this case because of missing rule inheritances.

In contrast to ETL, ATL applies the Map Objects before Links pattern in its execution strategy. An ATL rule such as

```
rule mapTransition
{ from s : Transition
 to t : Transition1
 ( name <- s.name,
  source <- s.source,
  target <- s.target )
}
```

is executed both in an initial execution phase, to create t and store an implicit trace of s mapping to t, and in a main execution phase, where the bindings of the *to* clause are executed using lookup of target elements created in the first phase. The assignment $source \leftarrow s.source$ is interpreted as "look up the target elements that correspond to the elements of $s.source$ and assign the collection of these elements to $t.source$".

QVT-R specifies transformations using mainly declarative relations, which act as transformation rules. Relations are distinguished as *top-level*: rules whose execution may commence without an explicit call, or non-top-level: rules which only execute if explicitly invoked from the `where` clause of a relation. No imperative statement language is provided, however imperative elements of OCL (the *iterate* operator) can be used. QVT-R operations are pure query functions, and cannot be used to modify the target model, in contrast to operations in the other MT languages.

The `when` clause in QVT-R enables one relation to look up elements produced by another relation, as with ATL's *resolveTemp*. It is an explicit data-dependency which introduces syntactic and

Table 8
Facilities of MT languages.

Aspect	*ATL*	*ETL*	*QVT-R*	*UML-RSDS*
Rule–rule dependencies	Explicit calls + *resolveTemp* references	Explicit + implicit calls	Explicit calls (`where`) + `when` references	None
Rule inheritance	Supported	Supported	Supported	Not supported
Rule-operation dependencies	Rules can call query and update operations	Rules can call query and update operations	Rules can call query operations or 'black box' updaters	Rules can call query/update operations
Operation-rule dependencies	Operations can use *resolveTemp*	Operations can use *equivalent*	None	None
Operation inheritance	Supported	Supported	Not supported	Supported
Update-in-place Execution	With restrictions	Supported	Supported; but unclear semantics	Supported
Transformation composition	Superposition (Import); Sequencing	Sequential composition (via script)	`extends` (Import)	Client–supplier; Import; Sequencing

semantic dependency of the calling relation upon the called relation. Additionally, it also acts as a guard, to prevent execution of the caller until the required elements have been produced by the referred relation. A clause

```
when { P(a,b) }
```

in relation R will be counted as a coupling between R and P. Unlike the use of *equivalent* in ETL, this mechanism does not invoke P to produce b from a.

The OCL concrete syntax used for QVT-R relation domains differs from that of the other MT languages. We evaluate complexity directly on this syntax, rather than upon its standard OCL translation. Thus a QVT-R object specification e

```
obj : E1 { att = var, rel = obj2 : E2{} }
```

has $c(e) = 11$, versus 19 for its conventional OCL equivalent expression:

$obj : E1 \ and \ obj.att = var \ and \ obj2 : E2 \ and \ obj.rel = obj2$

In QVT-R specifications, relations may define a large number of auxiliary local variables to transfer data from one relation domain to another, or to transfer data between relations (eg., the variables *var*, *obj*2 above). This specification style may result in high *EPL* values even for small transformations. This can cause problems in understanding the relations because the meaning and role of each variable needs to be understood. Such relations are also difficult to decompose using non-top relations. The issue is related to the strong support for bidirectionality in the language: it is the only one of the four languages to provide intrinsic support for bidirectional transformations (bx), and local variables facilitate bx specification.

UML-RSDS specifies transformations (and general applications) using UML and OCL 2.4. Transformations are expressed as use cases, the postcondition constraints of the use case serve as transformation rules. These are specified declaratively by OCL predicates. Unlike ATL, ETL and QVT-R, UML-RSDS rules cannot call or explicitly depend upon other rules, but only invoke operations of the transformation or of metamodel classes. This simplifies the call graph within the transformation, however self-recursive and mutually-recursive calling relations can exist between operations. In large specifications there is a tendency for most of the functionality of a transformation to be expressed using operations, which leads to high coupling. Operations and use cases can also be given imperative statements to define their effect, as with the procedural parts of ATL and ETL rules. Object lookup and target object resolution are performed declaratively using identity attributes.

Table 8 summarises the differences between the four languages regarding calling relationships and execution modes. In the final row we list the transformation composition facilities available in the languages. Composition mechanisms are one area where MT languages have poor support (Kusel et al., 2015). Apart from external sequencing using scripts, these mechanisms are rarely used. Although QVT-R provides an `extends` facility to combine transformations, this does not seem to be implemented by any QVT-R tools, and we found no use of this mechanism in the surveyed QVT-R cases.

5. Indicator analysis

We performed evaluation of the technical debt indicators on the individual transformations of each ATL, ETL, QVT-R and UML-RSDS case. Tables 44–48 in the Appendix show the measures of quality flaws for these individual transformations and for their containing cases. Underlined measures in Tables 44–48 identify where thresholds of the measures are exceeded, i.e., when specific flaws occur in individual transformations and cases. We present summaries of the results, for the cases, in Tables 9–12. Industrial cases are marked with a *.

For each language, we show in Tables 44–48 in the *ETS* column the size measures *rs* of the transformation rules and *os* of the helper operations, after their total. These are expressed in LOC for ATL, ETL and QVT-R, and as *c* values for UML-RSDS. The *ENR* column gives the number of rules in the case, *ENO* is the number of operations. The *ERS* column gives the number of rules with length over the threshold (50 LOC for ATL, ETL and QVT-R, c = 100 for UML-RSDS), likewise *EHS* for operations. The *EPL* column contains the number of rules/operations with more than 10 parameters, including local auxiliary variables. *EFO* is the number of rules/operations which have calling dependencies to more than 5 different rules/operations. *EFI* is the number of rules/operations which have more than 5 different client rules/operations. *CC* is the number of rules/operations over the *CC* threshold (10). *CBR* is expressed as $CBR_1(CBR_2)$ where CBR_1 is the total number of distinct calling dependencies, and CBR_2 is the number of rules/operations which occur in cycles of calling dependencies (they directly or transitively call themselves). *DC* is the number

Table 9
Technical debt indicator summary for ATL.

Transformation	*Category* Lucio et al. (2016)	*Style*	*LOC*	$c(\tau)$	*% in rules*	*# flaws*	*flaws/LOC*
Families2Persons	Migration (H)	F	39	89	38%	0	0
EliminateInherit	Refactoring	F	53	118	58%	2	0.038
Public2Private	Refactoring	L	79	213	96%	2	0.025
ReplaceAssoc	Refactoring	L	84	197	93%	2	0.024
Class2Relational	Refinement	L	97	245	97%	2	0.021
cdrat	Refactoring	H	105	274	100%	2	0.019
EMF2KM3	Migration (H)	L	118	227	92%	1	0.008
MergingPartial	Refactoring	H	185	491	97%	4	0.022
PartialRolesTotal	Refactoring	L	243	777	98%	5	0.021
AssertionModification	Refactoring	L	273	805	58%	7	0.026
SimpleClass2 SimpleRDBMS	Refinement	H	302	567	55%	10	0.033
ExcelExtractor	Code generation	F	311	528	81%	6	0.019
MDL2GMF	Refinement	F	384	815	45%	5	0.013
*KM32Measure**	Analysis	F	392	1015	57%	12	0.031
ExcelInjector	Migration (H)	F	395	601	58.5%	8	0.02
KM3toDOT	Refinement	F	451	926	55.6%	9	0.019
Make2Ant	Code generation	F	456	808	72%	6	0.013
FM2BPMN	Refinement	H	474	1241	75%	21	0.044
UML2MOF	Migration (H)	L	585	824	79.5%	8	0.013
MDL2UML21	Refinement	F	645	1004	74%	14	0.022
Monitor2Semaphore	Refinement	F	886	1643	89%	11	0.012
MOFtoUML	Migration (H)	F	935	1002	79.7%	18	0.019
MySQLtoKM3	Abstraction	F	995	1726	57.3%	20	0.02
Maven2Ant	Migration (H)	F	1307	3075	87%	18	0.014
PetriNetFrom/to	Semantic map	F	1604	1974	47.7%	29	0.018
Average			455.9	847.4	71%	8.88	0.0195

of distinct cloned expressions or statements e with $t(e) > 10$ in the case.

To compute the number of flaws in a transformation, we count 1 for each of *ETS*, *ENR*, *ENO*, *UEX*, CBR_1 over the thresholds (since these flaws are either present or absent for the transformation as a whole), plus *ERS* + *EHS* + *CC* + *EPL* + *EFO* + *EFI* + *DC* + CBR_2 (flaws which may have several occurrences within a transformation). For a transformation case which is a system of several transformations, we sum the number of flaws in each of its subtransformations. We normalise the number of flaws for different cases by dividing the flaw count by the transformation size (LOC or complexity c) to obtain the flaw density figures in Tables 9–12.

5.1. ATL

For ATL we consider the cases of Table 44 from the ATL transformations zoo and the *cdrat* case from Kolahdouz-Rahimi et al. (2014). For ATL, *UEX* is $n*(n-1)/2$ where n is the number of concrete non-lazy, non-called rules. This is the number of alternative orderings of n elements. Where a transformation consists of several subtransformations, we list these as (i), (ii) etc. in Table 44 below the main transformation entry. If a transformation uses an ATL source code library, then the technical debt of that library is also included in our analysis.

Table 9 gives a summary of the technical debt indicators of these cases. In summary tables the transformation category from Section 2.3 is shown (Migration (H) means a heterogeneous migration category, etc.). In the style column F denotes that the case uses the declarative functional style, H the hybrid style and L the declarative logical style of specification. The percentage of the specification contained in rules is also given, this is the sum of the rule sizes, divided by the total transformation size: $\frac{rs}{rs+os}$. The average flaw density for the ATL cases is 0.0195, i.e., there is approximately one flaw per 50 lines of specification text. It is noticeable that the number of flaws per LOC is quite similar across all of the cases (the variance is $8.4 * 10^{-5}$), even though the cases were written by a range of different authors and address a wide range of transformation problems. A scatter-plot of flaws against LOC is shown in Fig. 3, this also shows a high level of consistency across cases, with few outliers from the main trend line.

The ratio of complexity to LOC is 1.86, reflecting the relatively low semantic density of typical ATL specifications. The flaw rate per semantic element is 0.0105 (number of flaws divided by complexity c).

5.2. ETL

For ETL we define UEX as $\frac{n*(n-1)}{2}$ where n is the number of concrete non-lazy rules, excepting *pre* and *post* rules. This is the number of rules whose relative execution order is not fixed. Table 45 gives the measures for the selected ETL cases.

Table 10 gives a summary of the quality flaws of these cases. It is noticeable that the rate of flaws per LOC is higher (0.026) than for ATL in general (i.e., more than 1 flaw per 40 LOC), and with a much wider range of rates than for ATL (the variance is 0.0014). This may be due to the wide variety of styles supported by ETL, from the mainly imperative transformations of *StateElimination*, to the very implicit and declarative *CopyOO*. In the most complex cases, such as *MDDTIF*, three forms of inter-element dependence are used simultaneously: inheritance, explicit calls and implicit calls, leading to high values for *CBR*. Operations are used more extensively than in ATL, consisting of 55% of the specification text on average, compared to 29% in ATL. The scatter plot of flaws against LOC (Fig. 4) also shows the wide variation in flaw levels across ETL cases, with many outliers. Fig. 5 shows the estimated probability distribution functions of the ATL and ETL flaw density distributions.

The distribution of ETL flaw densities is strongly non-normal (normality is rejected at the 1% significance level by both the Anderson–Darling test (Anderson and Darling, 1952) and by the Shapiro–Wilks test (Shapiro and Wilk, 1965)), in contrast to the ATL flaw density distribution, which is consistent with the normal distribution at the 30% level (Shapiro–Wilks) and 10% level (Anderson–Darling). The ATL and ETL distributions are distinct at the 2% level using the Welch t-test (Weatherill, 1978) and at the 5% level using the Kolmogorov–Smirnov test (Smirnov, 1948) (Table 16).

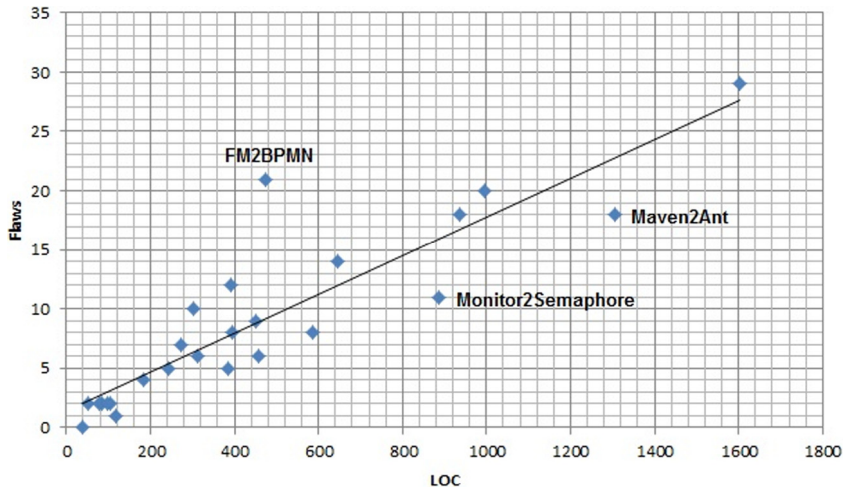

Fig. 3. ATL case flaws versus LOC.

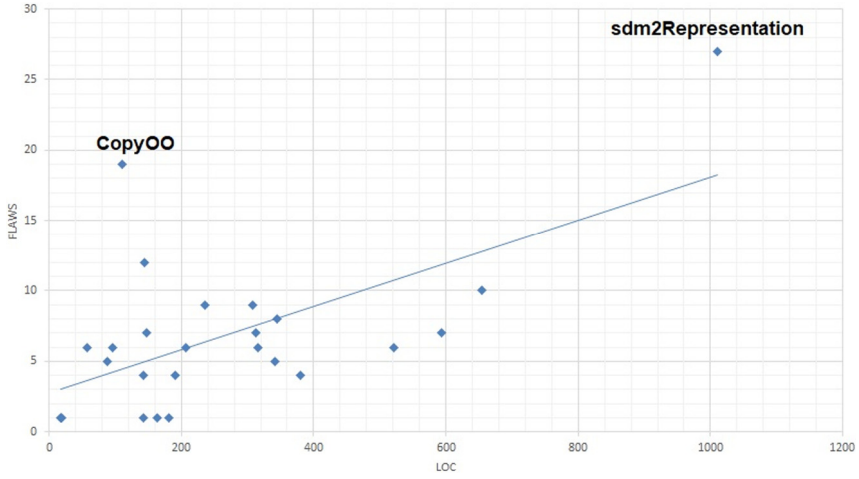

Fig. 4. ETL case flaws versus LOC.

Table 10
Technical debt indicator summary for ETL.

Transformation	*Category*	*Style*	*LOC*	$c(\tau)$	*% in rules*	*# flaws*	*flaws/LOC*
uml2xsd	Migration (H)	L	17	44	100%	1	0.059
In2out	Migration (H)	L	19	53	100%	1	0.052
CopyFlowchart	Migration (C)	L	57	177	100%	6	0.105
RSS2ATOM	Refinement	L	88	154	84%	5	0.057
Argouml2ecore	Migration (H)	H	96	321	79%	6	0.0625
CopyOO	Migration (C)	L	110	438	100%	19	0.173
OO2DB	Refinement	H	142	464	85.2%	1	0.007
CDO	Analysis	F	143	693	28%	4	0.028
MDDTIF	Refinement	H	145	377	95.8%	12	0.083
uml2Simulink	Refinement	H	148	477	77%	7	0.047
Flowchart2HTML	Code-generation	F	163	377	100%	1	0.006
QuickerMobile2Ionic	Migration (H)	H	181	501	29%	1	0.006
json2sql	Migration (H)	H	190	520	59%	4	0.021
TTC17Live	Refinement	H	206	573	79%	6	0.029
MarketToView	Code-generation	H	235	722	24%	9	0.038
BPMN2TBP	Migration (H)	H	308	645	100%	9	0.029
StateElimination	Refactoring	I	313	1062	49.5%	7	0.022
ECore2GenModel	Migration (H)	I	316	847	54%	6	0.019
SQL2Java	Migration (H)	I	341	1165	13.4%	5	0.015
cdrat	Refactoring	I	344	951	100%	8	0.023
JEE	Refinement	I	380	814	16.5%	4	0.011
AntlrAst2Intermediate	Abstraction	F	522	1124	22%	6	0.0115
Newsletter2Android	Code-generation	I	593	1321	73%	7	0.012
Java2Html	Code-generation	I	654	2693	2%	10	0.015
sdm2Representation	Refinement	I	1011	3882	9%	27	0.027
Average			269	816	45%	6.88	0.026

Table 11
Technical debt indicator summary for QVT-R.

Transformation	*Category*	*Style*	*LOC*	$c(\tau)$	*% in rules*	*# flaws*	*flaws/LOC*
hsm2nhsm (recursion)	Abstraction	F	48	105	100%	2	0.042
ClassModelToClassModel	Migration (E)	L	85	85	100%	2	0.0235
HSM2FlatSM	Abstraction	L	85	137	93%	1	0.012
UmlToRel	Bidirectional	L	98	75	66%	0	0
SeqToStm	Refinement	L	104	175	100%	1	0.009
pn2pnw	Bidirectional	L	115	170	100%	2	0.017
set2oset	Bidirectional	F	121	120	91%	2	0.0165
SeqToStmc	Refinement	F	149	162	100%	4	0.027
bag12bag2/bag22bag1	Bidirectional	F	157	151	97%	1	0.006
ER2WebML/WebML2ER	Bidirectional	L	190	508	83%	1	0.005
Ecore2copyQVT	Higher-order	L	193	591	100%	7	0.036
cdrat	Refactoring	F	202	701	100%	6	0.03
UmlToRdbms	Refinement	F	238	314	95%	6	0.025
DNF_bbox	Refactoring	L	263	470	100%	11	0.042
gantt2cpm	Bidirectional	F	378	571	87%	10	0.026
DNF	Refactoring	L	396	665	100%	16	0.04
families2persons	Bidirectional	H	435	687	92%	17	0.039
dag2ast/ast2dag	Bidirectional	F	439	564	78%	8	0.018
f2p/p2f	Bidirectional	H	462	822	72%	12	0.026
Bpmn2UseCase	Migration (H)	F	532	1131	100%	17	0.032
ecore2sql2	Bidirectional	F	956	1428	84%	34	0.036
ecore2sql3	Bidirectional	F	960	1453	80%	31	0.032
communication2class	Refinement	F	1029	1122	38%	15	0.0145
ecore2sql1	Bidirectional	F	1120	1813	70%	42	0.038
*RelToCore**	Refinement	H	2038	5415	95%	63	0.031
Average			431.7	777.4	83%	12.4	0.029

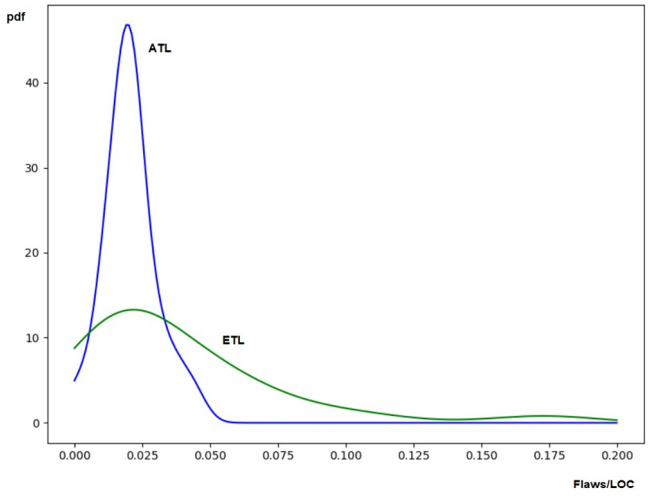

Fig. 5. ATL and ETL flaw density distributions.

From Table 10 we have that complexity/LOC for ETL is 3.03, indicating a greater semantic density in ETL specifications than for ATL. The rate of flaws per semantic element for ETL is 0.008, similar to ATL.

5.3. QVT-R

For QVT-R transformations the *CBR*, *EFO* and *EFI* indicators are of particular interest, since QVT-R rules (termed 'relations') may be interdependent in several different ways: a rule may refer to another in its `when` or `where` clause, and may have a recursive dependency upon itself, and may override another rule. *UEX* is taken as $\frac{n*(n-1)}{2} - W$ where n is the number of concrete top-level rules in a transformation and W is the number of enforced orderings defined by `when` predicates $P(args)$ in top-relations R requiring that top relation P has been executed prior to R.

Table 46 gives the measures for the selected QVT-R cases. Table 11 gives a summary of the technical debt of these cases. There are 0.029 flaws/LOC and 0.016 flaws per semantic element, figures higher than for ATL or ETL. There are 1.8 semantic elements/LOC, a density figure similar to ATL.

Fig. 6 shows the results graphically. Apart from one transformation with an unusual functional structure (*communication2class*) and a highly complex bx (*ecore2sql1*), there is high correlation between LOC and the number of flaws.

5.4. UML-RSDS

For UML-RSDS transformations we consider four substantial case studies: the two parts of the UML2C code generator (Lano et al., 2017), the UML to Python code generator *uml2py*, and the class diagram modulariser *cra* from Lano et al. (2016). A range of other examples are also included from nms.kcl.ac.uk/kevin.lano/zoo. For UML2C and *uml2py* we examine several versions, and for *cra* we consider the original (2016) and revised (2017) versions (Table 48).

In total there are 57 individual transformations and 25 transformation systems. Tables 47 and 48 show the measures for these transformations and their individual subtransformations. *UEX* is not shown because this is always 0 in UML-RSDS transformations. *EFI* is also 0 in these cases. For *ETS*, *ERS* and *EHS* we use $c()$ measures and thresholds of 1000 for transformations and 100 for rules and operations.

Table 12 summarises the results for UML-RSDS. The percentage of a transformation consisting of rules is measured wrt c. It can be seen that there is a substantial difference in this proportion for UML-RSDS (25%) compared with the other languages, which all have over 45% of their content in rules. Fig. 7 shows the results graphically.

Excessive use of helpers produces transformations which are akin to programmes in a functional programming language. In the largest transformations (*uml2Cb*, *uml2py*, *cra*) there is a considerable imbalance of functionality towards helpers, whilst smaller transformations such as the Monte-Carlo simulator are more balanced.

There are 0.037 flaws/LOC, the highest figure of any of the languages. However, if the two largest cases are excluded, the ratio becomes 0.027, similar to QVT-R. The flaws/c ratio is 0.013

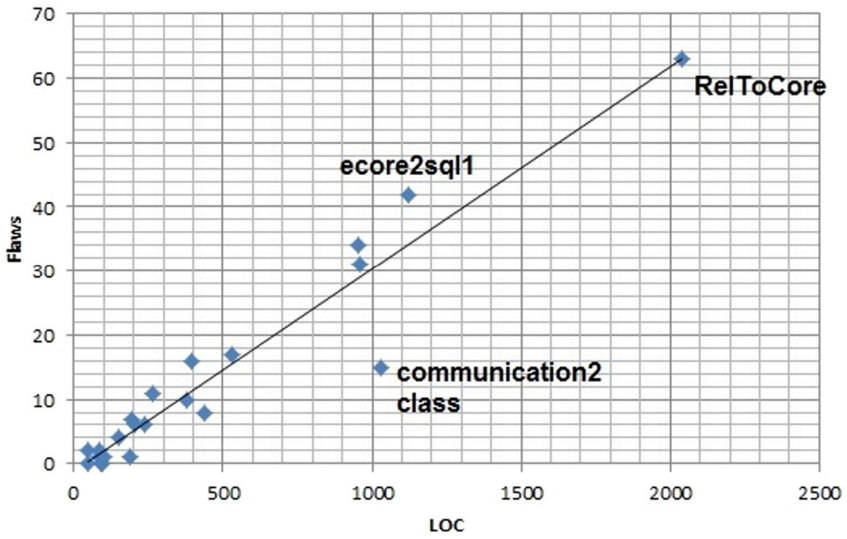

Fig. 6. QVT-R case flaws versus LOC.

Table 12
Technical debt indicator summary for UML-RSDS.

Transformation	*Category*	*Style*	*LOC*	$c(\tau)$	*% in rules*	*# flaws*	*flaws/LOC*
Tree2Graph	Refinement	L	4	28	100%	0	0
calc	Analysis	L	15	83	100%	0	0
cdrat	Refactoring	L	26	222	100%	1	0.038
GMF migration	Migration (E)	H	38	116	29%	0	0
Monte-Carlo sim	Analysis	L	51	90	68%	0	0
f2p/p2f	Bidirectional	L	58	158	86%	3	0.052
*CDO**	Analysis	F	94	182	17%	2	0.021
uml2rdb	Refactoring	L	102	184	84%	1	0.009
PetriNet to SM	Refactoring	L	117	425	100%	1	0.008
movies	Analysis	F	156	432	40%	3	0.019
TTC14Live	Reactive	F	192	999	15%	7	0.036
ATL2UMLRSDS	Migration (H)	F	210	593	30%	6	0.028
ActivityMigration	Migration (E)	L	217	935	2.2%	9	0.041
*Simplex**	Analysis	F	294	664	1%	4	0.014
QVT2UMLRSDS	Migration (H)	F	315	803	9%	10	0.032
bootstrap	Analysis	F	338	574	39%	5	0.015
StatLib	Analysis	F	370	525	9%	2	0.005
*Nelson–Seigal**	Refinement	H	458	1219	67%	15	0.033
cra (2016)	Refactoring	H	490	1360	32%	12	0.024
uml2Ca V0.3	Code generation	L	569	842	87%	8	0.014
uml2py V0.4	Code generation	F	605	3157	6%	50	0.083
cra (2017)	Refactoring	H	844	1635	45%	12	0.014
uml2Ca V0.9	Code generation	L	874	1272	69%	22	0.025
uml2Cb V0.9	Code generation	F	1576	5621	16%	119	0.075
uml2py V0.5	Code generation	F	1628	4846	2%	69	0.042
Average			385.6	1078.6	25%	14.4	0.037
Average excluding *uml2CbV*0.9, *uml2pyV*0.5			279.9	717.3		7.52	0.027

taken over all 25 cases, and 0.01 for the reduced subset. The $c(\tau)$ measure is around 2.8 times the LOC for the complete set of cases, similar to ETL.

Fig. 8 shows the probability density function of the QVT-R and UML-RSDS flaw density distributions. The QVT-R distribution is consistent with the normal distribution at the 10% level (Shapiro–Wilks test) and 15% level (Anderson–Darling test) whilst the UML-RSDS distribution is inconsistent with normality at the 2% level (Shapiro–Wilks test) and 15% level (Anderson–Darling).

6. Discussion and summary of results

In this section we consider the detailed results for each language with respect to the four research questions:

RQ1: What is the extent of technical debt in MT cases?

RQ2: What are the most frequent technical debt issues in MT cases?

RQ3: How does the level and character of TD vary between MT languages and between different MT application categories?

RQ4: What are the differences between the extent and character of TD in MT languages and in traditional programming languages?

6.1. RQ1: Quality flaw prevalence

For ATL, for **RQ1**, 36 of the 38 individual transformations had flaws (95%), and 24 of the 25 transformation cases contained transformations with flaws (96%).

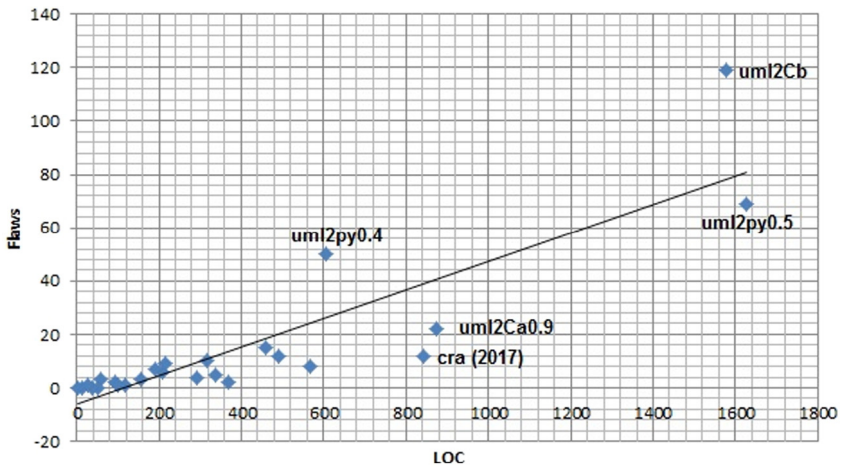

Fig. 7. UML-RSDS case flaws versus LOC.

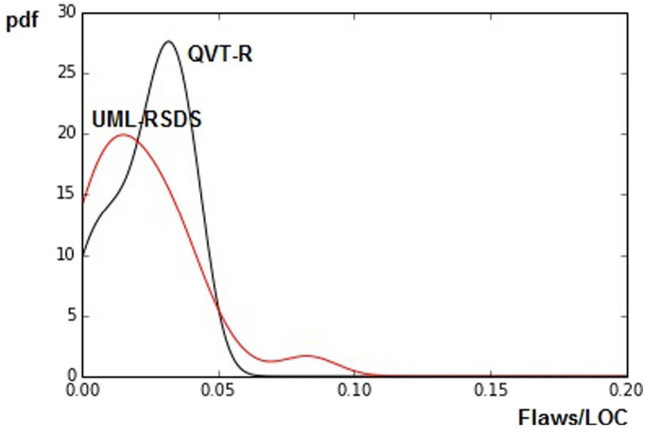

Fig. 8. QVT-R and UML-RSDS flaw density distributions.

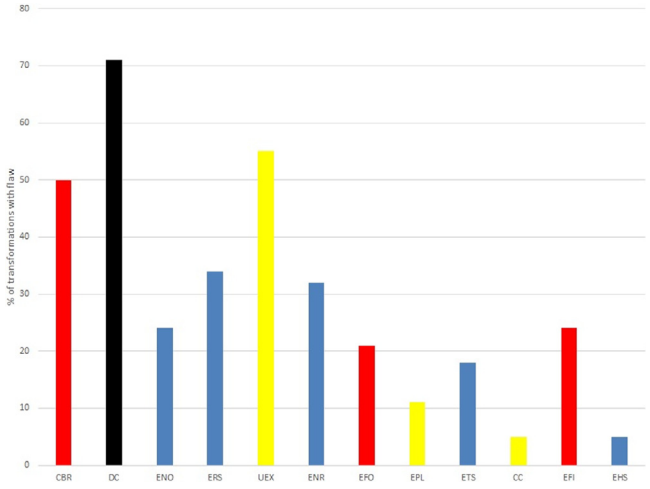

Fig. 9. Flaw frequencies in ATL.

Table 13
Prevalence of quality flaws.

Language	*% transformations with flaws*	*% cases with flaws*
ATL	95%	96%
ETL	82%	100%
QVT-R	96%	96%
UML-RSDS	59%	84%
Overall	79%	94%

For ETL, for **RQ1**, 31 of the 38 individual transformations contained flaws (82%), and all of the 25 transformation cases contained transformations with flaws (100%).

For QVT-R, for **RQ1**, out of 25 transformations and cases, 24 had flaws (96%).

For UML-RSDS, for **RQ1**, out of 57 transformations, 34 had some flaws (59%), whilst 21 of 25 transformation cases contained some transformations with flaws (84%).

To summarise for **RQ1**, we find that overall 125/158 individual transformations contain quality flaws which can be technical debt indicators (79% of transformations), whilst 94 of 100 cases (94%) have such flaws (Table 13). This shows that the quality flaws are pervasive in the selected transformation cases.

6.2. RQ2: Relative frequencies of quality flaws

For ATL, wrt **RQ2**, the most common quality flaws were *DC* > 0 (27/38), CBR – either $CBR_1 > nrules + nops$ or $CBR_2 > 0$ – (19/38), *UEX* (21/38), *ERS* (13/38), *ENR* (12/38), *ENO* (9/38), *EFI* (9/38) and *EFO* (8/38). The percentages of ATL transformations with each flaw are shown in Fig. 9. Blue coding indicates size category flaws (ENO, ENR, ETS, ERS, EHS), yellow coding shows semantic complexity flaws (UEX, EPL, CC), red coding is for dependency flaws (CBR, EFO, EFI) and black for redundancy flaws (DC).

For ETL wrt **RQ2** the most common flaws were *CBR* (20/38), *DC* (15/38), *ERS* (8/38) and *UEX* (7/38). The percentages of ETL transformations with each flaw are shown in Fig. 10.

For QVT-R wrt **RQ2**, *CBR* occurs in 20 of 25 transformations, *EPL* in 14, *DC* in 13 and *ERS* in 12. High values of *EPL* arise because of the use of many local variables within QVT-R relations. *CBR* flaws arise from the unstructured nature of QVT-R transformations, in which rules may be highly inter-dependent. Fig. 11 shows the frequencies of different types of flaw in the QVT-R cases.

For UML-RSDS wrt **RQ2**, excessive *CBR* occurs in 22 cases. *DC* occurs in 21 cases. *ENO* occurs in 20 cases. *CC* occurs in 11 cases. In these cases, the coupling issues concern complex dependencies between helpers, rather than between rules. The prevalence of *CBR* and *ENO* flaws suggest overuse of helpers/operations. Poor structure and high numbers of flaws were apparent in the largest transformations, even in cases such as *uml2py* where transformation reuse (of *uml2Ca* and *uml2Cb*) took place. Fig. 12

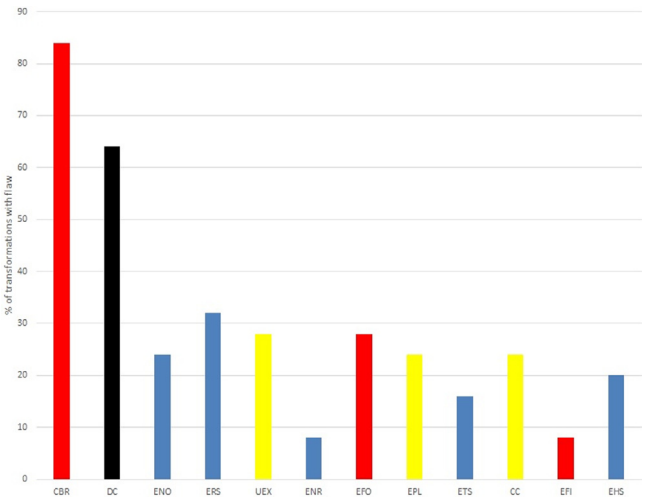

Fig. 10. Flaw frequencies in ETL.

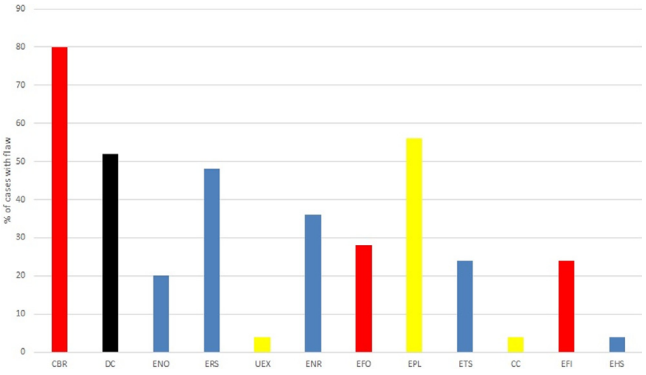

Fig. 11. Flaw frequencies in QVT-R.

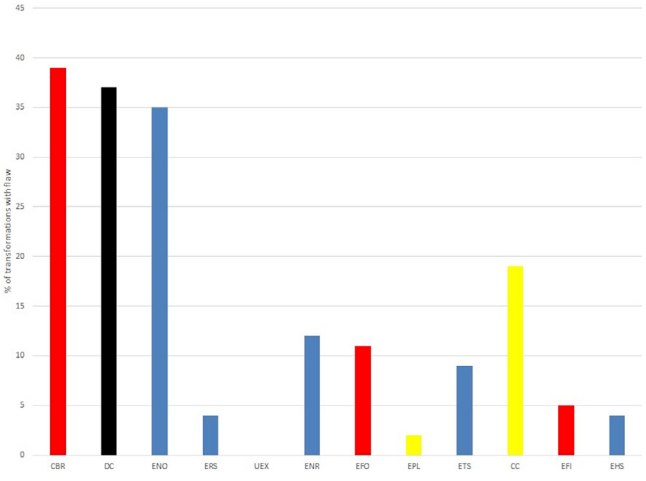

Fig. 12. Flaw frequencies in UML-RSDS.

shows the frequency of different flaw categories in UML-RSDS transformations.

Fig. 13 shows the total number of cases with each type of flaw. In summary for **RQ2**, we find that excessive *CBR* and *DC* are clearly the most frequently-occurring quality flaws which arise across all MT languages: these flaws each occur in approximately 50% of all the surveyed transformations. No other flaw category has a frequency of occurrence above 25%.

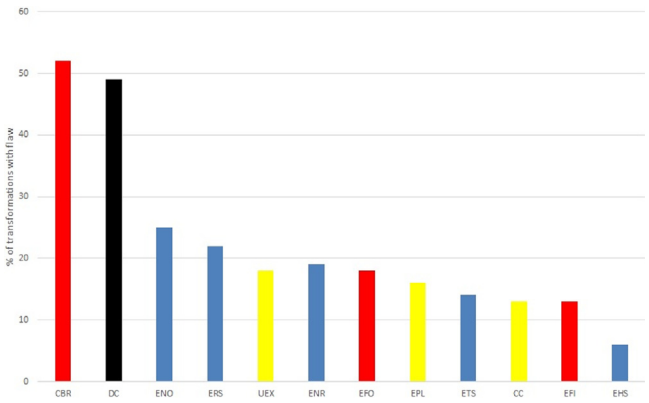

Fig. 13. Flaw categories and frequencies for all languages.

Table 14
Quality flaw frequency in different MT languages.

Flaw category	*ATL*	*ETL*	*QVT-R*	*UML-RSDS*	*Overall*
CBR	19/38	21/38	20/25	22/57	82/158
DC	***27/38***	16/38	13/25	21/57	77/158
ENO	9/38	6/38	5/25	***20/57***	40/158
ERS	13/38	8/38	***12/25***	2/57	35/158
ENR	12/38	2/38	9/25	7/57	30/158
UEX	***21/38***	7/38	1/25	0/57	29/158
EFO	8/38	7/38	7/25	6/57	28/158
EPL	4/38	6/38	***14/25***	1/57	25/158
ETS	7/38	4/38	6/25	5/57	22/158
EFI	9/38	2/38	6/21	3/57	20/158
CC	2/38	6/38	1/25	***11/57***	20/158
EHS	2/38	5/38	1/25	2/57	10/158

6.3. RQ3: Differences in quality flaw frequencies between languages

With regard to the first part of **RQ3**, Table 14 summarises the prevalence of different quality flaws in different MT languages, counting the number of individual transformations which have flaws of each kind. Unusual patterns in the frequencies of flaws are emphasised. There are significant variations in the relative frequency of quality flaws in different languages, with DC more common than CBR in the case of ATL, and a relatively high frequency of EPL and ERS in the case of QVT-R, and of ENO and CC in the case of UML-RSDS.

In Tables 13 and 14 it is noticeable that the dispersion of flaws across transformations appears to be wider for ATL and QVT-R compared to the other languages. Considering the top 3 most-common flaws in each language, we find that these occur in 95% of the ATL transformations, and 92% of QVT-R transformations, compared to 82% for ETL and 56% for UML-RSDS. This indicates that these flaws occur quite uniformly across the ATL and QVT-R cases, whilst at the other extreme, 52% of the UML-RSDS flaws occur in just 2 of the cases.

Table 15 compares the overall figures for size and flaw frequencies, for each language.

The flaw density figures for QVT-R and UML-RSDS are higher than for ATL and ETL, both wrt LOC and wrt *c*. This difference can be due to specific language features such as excessive use of auxiliary variables (QVT-R), or excessive use of operations (UML-RSDS), but also due to the use of UML-RSDS for more complex transformations, including update-in-place cases such as *PetriNet to SM* which would be very difficult to express in ATL, and the use of QVT-R for bx. Large-scale code-generation transformations are particularly prone to flaws in UML-RSDS. If the two largest cases were excluded from the UML-RSDS data, then UML-RSDS would

Table 15
Overall size and quality flaw results.

Language	LOC	*c*	*c*/LOC	Flaws	Flaws/ LOC	Flaws/ *c*
ATL	11,398	21,185	1.86	222	0.0195	0.0105
ETL	6722	20,395	3.03	172	0.026	0.008
QVT-R	10,793	19,435	1.8	311	0.029	0.016
UML-RSDS	9641	26,965	2.8	361	0.037	0.013
Overall	38,554	87,980	2.28	1066	0.0276	0.0121

Table 16
Language flaw distribution differences.

	ATL	*ETL*	*QVT-R*	*UML-RSDS*
ATL	–	$p < 0.05$	$p > 0.15$	$p > 0.5$
ETL	0.039	–	$p < 0.1$	$p < 0.05$
QVT-R	0.039	0.077	–	$p > 0.5$
UML-RSDS	0.3	0.27	0.257	–

Table 17
Language flaw distribution variance/skew/normality.

	ATL	*ETL*	*QVT-R*	*UML-RSDS*
Variance	$8.4 * 10^{-5}$	0.0014	$1.5 * 10^{-4}$	$3.71 * 10^{-4}$
Skew	0.057	0.31	−0.401	0.167
Anderson–Darling	$10\% < p < 15\%$	$p < 1\%$	$p \geq 15\%$	$10\% < p < 15\%$
Shapiro–Wilks	0.3	$p < 1\%$	0.24	0.0167

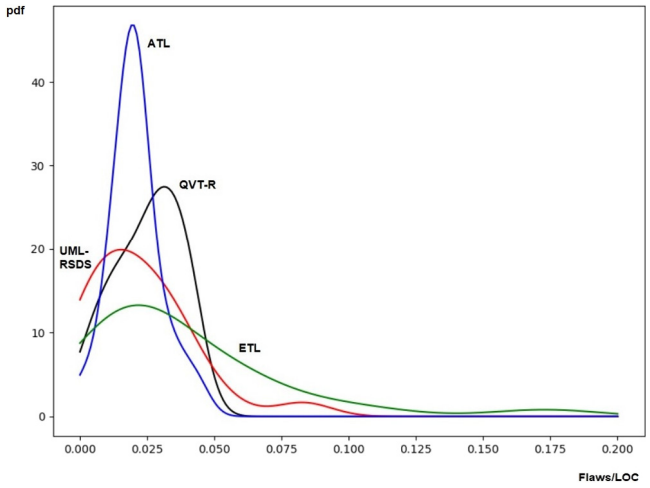

Fig. 14. Flaw density distributions of languages.

Table 18
Correlations of size/complexity and flaws.

Correlation	*ATL*	*ETL*	*QVT-R*	*UML-RSDS*
flaws with LOC	0.89	0.60	0.96	0.87
flaws with $c(\tau)$	0.80	0.67	0.92	0.97
flaws/LOC with LOC	−0.14	−0.41	0.30	0.49
LOC with $c(\tau)$	0.9	0.95	0.94	0.93

have flaws/LOC equal to 0.027 and flaws/*c* equal to 0.01. UML-RSDS and ETL are differentiated from ATL and QVT-R by having c/LOC ratios close to 3, whilst ATL and QVT-R have ratios closer to 2. Considering flaws/c, the differences between MT languages is less pronounced than for flaws/LOC, however there is still a gap between the flaw density of ATL and ETL and that of QVT-R and UML-RSDS.

Comparing the distributions of flaw densities (Figs. 5 and 8), we propose the null hypothesis $H_{L1,L2}$ that language $L1$ and language $L2$ have the same distribution of flaw densities, for $L1 \neq L2$, and test this using the Welch t-test and Kolmogorov–Smirnov tests. The upper triangle of Table 16 shows p values for the Welch t-test results of $H_{L1,L2}$, with distribution differences significant at the 5% level underlined. The lower triangle of the table shows p values for the Kolmogorov–Smirnov test. This shows there is evidence against $H_{ATL,ETL}$ at the 5% level and against $H_{ETL,QVT-R}$ at the 10% level, and no clear evidence against $H_{L1,L2}$ for other language combinations.

Table 17 shows the differences between the flaw density distributions in terms of their variances and skews, and the p-values for tests of normality of the distributions. Tests showing a significant difference from normality at the 5% level are underlined. This comparison shows that ETL is anomalous in terms of its high variance of flaw densities, and strong departure from normality, whilst QVT-R is anomalous in having a negative skew (i.e., a tail to the left instead of to the right). Fig. 14 shows the four distributions together.

It is interesting to consider if the number of flaws, or the flaw density, is directly related to transformation size, measured by LOC or c. The Pearson correlation coefficients between LOC and number of flaws, LOC and flaw density, and between $c(\tau)$ and number of flaws, are given in Table 18. High correlations are underlined, these are significant at the 5% level.

Generally, both LOC and c are good predictors of the number of flaws, for ATL, QVT-R and UML-RSDS. The relationship is shown visually for LOC on Figs. 3, 6 and 7. But for ETL there is only weak correlation of LOC and c with the number of flaws (Fig. 4), which shows that transformation size/complexity has low influence on the number of flaws for ETL specifications, in contrast to the case for the other languages. There is negligible relation between LOC and flaw density for ATL and QVT-R. For ETL, larger transformations tend to have lower flaw density, but for UML-RSDS there is a weak positive correlation of size with flaw density.

It should be noted that the correlation between c and LOC is high for all the languages, although the c/LOC ratio is different for different languages. This indicates a high degree of consistency in the c/LOC ratio between different cases of the same language.

To answer the second part of **RQ3** we can also compare the levels of quality flaws in different categories of transformation, across languages. Table 19 shows the flaw frequencies for the main categories of transformations in our survey. The difference in flaw levels between the categories are small, but are in general accord with expectations that more complex MT tasks such as refinement and bx will result in transformations with higher numbers of flaws compared to simpler tasks such as migration or abstraction.

Code generators are impacted by flaws of excessive numbers of operations/rules (due to the need to define a mapping for each category of expression or statement in the source language being processed — such as UML) and excessive CBR due to defining operation/rule dependencies that follow the mutually-recursive inter-dependencies between source language (eg., UML) elements. Bx have additional complexity over unidirectional transformations, caused by the need to support execution in two directions (Westfechtel, 2018a,b; Westfechtel and Buchmann, 2019).

For the four main categories of refinements, refactorings, migrations and code generations, we can further investigate if there are distinctions in flaw levels in each transformation category between different languages (Table 20). We indicate cases where the flaw level is at least 25% above or below the average for that language in general.

It is notable that QVT-R has poor flaw densities for refactoring cases, compared to ETL and UML-RSDS. For refactoring cases, ETL and UML-RSDS benefit from the additional capabilities of their

Table 19
Quality flaws for different MT categories.

Category	Cases	LOC	Flaws	Flaws/LOC
Code generation	11	7664	307	0.04
Bidirectional	13	5489	163	0.029
Refinement	21	9379	238	0.025
Refactoring	17	4119	96	0.023
Migration	22	6411	149	0.023
Abstraction	4	1650	29	0.018
Analysis	9	1853	32	0.017

Table 20
Quality flaws for different MT categories and languages.

Category	*Language*	LOC	Flaws	Flaws/LOC
Refactoring	ATL	1022	24	0.023
	ETL	657	15	0.022
	QVT-R	861	33	0.038 (> average)
	UML-RSDS	1579	27	0.017 (< average)
Refinement	ATL	3239	72	0.022
	ETL	2035	63	0.031
	QVT-R	3558	89	0.025
	UML-RSDS	462	15	0.032
Migration	ATL	3379	53	0.0157
	ETL	1635	58	0.035 (> average)
	QVT-R	617	19	0.03
	UML-RSDS	780	19	0.024 (< average)
Code-generation	ATL	767	12	0.0156
	ETL	1645	27	0.016 (< average)
	UML-RSDS	5252	268	0.05 (> average)

Table 21
Flaw density differences for different ETL styles.

Style	LOC	*c*	flaws	flaws/LOC	flaws/*c*
L	291	866	32	0.11	0.04
F	828	2194	11	0.01	0.005
H	1358	4600	54	0.04	0.01
I	3952	12,735	74	0.02	0.006

specification languages compared to QVT-R and ATL. For refinements and migrations, ETL has higher levels of flaws compared to its average and to the other languages, this seems to result from high numbers of coupling flaws. For example, 17 out of the 21 transformations/subtransformations of ETL migration or refinement cases have *CBR* flaws, whilst only 4 of the other 17 ETL individual transformations have such flaws (Table 45). For code-generation cases, UML-RSDS is also affected by high numbers of *CBR* flaws.

Regarding the influence of specification style on flaw density, we found small effects for ATL and QVT-R, however for ETL there are significantly fewer flaws for the functional and imperative styles, compared to the logical and hybrid styles (Table 21).

It is interesting to note that CC and CBR flaws are mutually exclusive for ETL cases — logical and functional styles of ETL transformation are based on implicit or explicit calls, and tend to have CBR flaws, whilst imperative cases are based on extensive procedural code and tend to have CC flaws. We observe that larger ETL cases tend to be written in more imperative styles, and that this is actually beneficial for overall flaw density levels, while increasing semantic complexity and size flaws relative to dependency flaws.

For UML-RSDS there are also substantial differences between styles, with the functional style more than twice as flaw-prone as the logical or hybrid styles (Table 22). We observe that the larger UML-RSDS cases tend to be written in a functional style, and that this leads to high numbers of flaws of all kinds.

Table 22
Flaw density differences for different UML-RSDS styles.

Style	LOC	flaws	flaws/LOC
L	2033	45	0.022
F	5778	277	0.048
H	1830	39	0.021

6.4. RQ3: Comparisons on common case studies

We can further investigate differences between the MT languages by considering common cases which are specified in several languages. Two case studies (class diagram restructuring (Kolahdouz-Rahimi et al., 2014) and UML to relational databases) have been carried out in all four languages, permitting an alternative comparison of flaw levels in these languages. For such cases, the number of flaws is more significant than the flaw density, since the different versions address the same problem.

The class diagram restructuring case is described in Kolahdouz-Rahimi et al. (2014). This is a refactoring transformation, and the developments were carried out by independent teams in order to compare the effectiveness of different MT languages on a complex refactoring problem. Table 23 compares quality flaws in the different language versions of this transformation. Note that the ATL version does not cover all the functionality because of the difficulty of expressing refactoring transformations in ATL. The LOC values differ slightly from Kolahdouz-Rahimi et al. (2014) because here we are including comment lines in LOC, and we have also expanded the QVT-R solution to cover all the required functionality. Our complexity measure *c* is more inclusive than that used in the original paper.

The style of specification is very different in each case, with the UML-RSDS version using a very dense logical specification with strict sequencing of steps, the ATL, ETL and QVT-R versions employ a 'refactoring by selective copy' strategy, but with imperative programming used in ATL and ETL, and a functional style in the QVT-R case. The ETL solution uses explicit object deletion, as in the UML-RSDS solution, and modularises the solution into three separate subtransformations. The analysis of these versions is summarised in Table 24.

The UML-RSDS solution has the smallest size and complexity, and the lowest number of flaws, whilst ATL has the lowest flaw density wrt LOC. The figures for flaws/c are 0.004 for UML-RSDS, 0.008 for ETL and QVT-R, and 0.007 for ATL.

Another common example is the classic UML to relational database refinement transformation. The cases for different languages have been independently developed at different times by different teams, usually as an illustrative example for each MT language. Table 25 lists the versions of this transformation in the different languages. The transformations use the "one table per class" mapping from UML to RDB: each persistent class has a corresponding table, and foreign keys are used to represent generalisation and association relations. There are two versions in ATL and two versions in QVT-R, with the second version in each case being an improved version of the first, however the ATL (2) version does not handle inheritance.

Table 26 gives a summary of the technical debt indicators of these cases. Each version is quite different in style. The UML-RSDS version decomposes the transformation into smaller sequential steps (*introPrimaryKeys*, *inheritance2ForeignKey*, *manymany2Table*, *onemany2Fk*), this facilitates reuse of parts of the transformation for alternative UML to RDB mapping strategies. It also uses an update-in-place refactoring approach to perform the refinement. The other solutions have a monolithic structure and use separate source and target models. The QVT-R (2) solution has the lowest

Table 23
Class diagram restructuring versions.

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EPL*	*EFO*	*CC*	*CBR*	*DC*	*UEX*
ATL	105 (105, 0)	6	0	1	0	0	0	0	1(0)	1	10
ETL	344 (344, 0)	16	0	2	0	1	0	1	16(0)	2	26
QVT-R	202 (202,0)	17	0	0	0	0	0	0	19(2)	2	0
UML-RSDS	26 (26,0)	3	0	0	0	0	0	0	0(0)	1	0

Table 24
Class diagram restructuring summary.

Transformation	*Category*	*Style*	*LOC*	$c(\tau)$	*% in rules*	*# flaws*	*flaws/LOC*
ATL	Refactoring	H	105	274	100%	2	0.019
ETL	Refactoring	H	344	951	100%	8	0.023
QVT-R	Refactoring	F	202	701	100%	6	0.03
UML-RSDS	Refactoring	L	26	222	100%	1	0.038

Table 25
UML to relational database versions.

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EPL*	*EFO*	*EFI*	*CBR*	*DC*	*UEX*
ATL (1)	302 (166, 136)	1	6	1	0	1	1	0	13(6)	0	0
ATL (2)	97 (94, 3)	6	1	0	0	0	0	0	6(0)	1	15
ETL	142 (121, 21)	6	3	0	0	0	0	0	8(1)	0	6
QVT-R (1)	238 (226,12)	7	1	1	0	1	0	0	10(3)	0	0
QVT-R (2)	92 (92,0)	7	0	0	0	0	0	0	7(0)	0	7
UML-RSDS	102 (94,8)	5	1	0	0	0	0	0	5(0)	1	0

Table 26
UML to relational database summary.

Transformation	*Category*	*Style*	*LOC*	$c(\tau)$	*% in rules*	*# flaws*	*flaws/LOC*
ATL (1)	Refinement	H	302	567	55%	10	0.033
ATL (2)	Refinement	L	97	245	97%	2	0.021
ETL	Refinement	H	142	464	85.2%	1	0.007
QVT-R (1)	Refinement	F	238	314	95%	6	0.025
QVT-R (2)	Bidirectional	L	92	292	100%	0	0
UML-RSDS	Refactoring	L	102	184	84%	1	0.009

number of flaws and flaw density. Apart from this, the ETL version has the lowest flaw density, wrt LOC and complexity.

Two other case studies are common to ETL, QVT-R and UML-RSDS: the CDO financial analysis application, which was additionally developed in QVT-R, and the tree-to-graph refinement transformation:

- *CDO*: in this case the ETL and UML-RSDS specifications were written by different teams. The original UML-RSDS version was written for an industrial collaborator, the ETL version was written for the comparative study of Lano and Alfraihi (2018), and was based on the UML-RSDS version. The UML-RSDS version has the least flaws, whilst the flaw density figures of the ETL and UML-RSDS versions are similar. High flaw density in the QVT-R version is caused by extensive use of recursion (6 CBR_2 flaws).
- *Tree2Graph*: the original version is in ETL (Kolovos et al., 2008). The QVT-R and UML-RSDS versions were written for this paper. The ETL and QVT-R versions are larger in both LOC and complexity compared to the UML-RSDS version, and the ETL version has a higher number of flaws and higher flaw density than the other two solutions.

Table 27 summarises the results for these cases.

Considering flaw counts, UML-RSDS has a total of 2 flaws for the first two common case studies, ETL has 9, while the best ATL and QVT-R versions have totals of 4 and 6 respectively. For the CDO/Tree2Graph case studies, UML-RSDS has 2 flaws, QVT-R 8 and ETL 5. This shows that UML-RSDS can produce good quality solutions in these relatively small cases.

In summary for **RQ3**, we have identified significant differences in the level and types of quality flaw indicators for TD between the languages. Contrary to expectation, we found higher TD levels in the more declarative languages QVT-R and UML-RSDS, and this tendency is particularly evident in the largest cases for these languages.

6.5. RQ4: Model transformation technical debt compared to code TD

For **RQ4**, TD levels in developer-coded Eclipse projects have been estimated in He et al. (2016) using similar flaw indicators to ours. The projects had flaw density values ranging from 0.005 to 0.04 flaws per LOC, with an average around 0.015. We also evaluated manually-coded versions of a UML to C++ translator (18,100 lines of Java), and 2 versions of the CDO case study (163 lines of C++, and 196 lines of Java) (Alfraihi and Lano, 2018; Lano and Alfraihi, 2018) using the PMD code analyser (https://pmd.github.io). These had flaw levels of 0.009/LOC, 0.024/LOC and 0.02/LOC, respectively. The flaw levels of QVT-R and UML-RSDS are high in comparison with these code results, whilst ATL and ETL exhibit flaw levels more typical of executable code. It is interesting to note that the most procedural of the four languages (ETL) also has the lowest flaw density wrt c. We have also evaluated 40 cases in the procedural QVT-O (OMG, 2016), EGL (Epsilon Generator Language, 2020) and Kermeta (Anon, 2020) MT languages using the same TD indicators as this paper (Kolahdouz-Rahimi et al., 2020a), and identified flaw density levels of 0.016 flaws/LOC for QVT-O, 0.019 for EGL, and 0.015 flaws/LOC for Kermeta. These are also similar to code flaw density levels.

Table 27
ETL and QVT-R compared to UML-RSDS on common cases.

Transformation	*Category*	*Style*	*LOC*	$c(\tau)$	*% in rules*	*# flaws*	*flaws/LOC*
CDO (ETL)	Analysis	F	143	693	28%	4	0.028
CDO (QVT-R)	Analysis	F	92	274	8%	8	0.087
CDO (UML-RSDS)	Analysis	F	94	182	17%	2	0.021
Tree2Graph (ETL)	Refinement	L	15	37	100%	1	0.067
Tree2Graph (QVT-R)	Refinement	L	17	40	100%	0	0
Tree2Graph (UML-RSDS)	Refinement	L	4	28	100%	0	0

High flaw levels in MT languages may arise in part because MT languages are still somewhat experimental and immature compared to programming languages. MT language features and semantics are more difficult to understand and to use than more conventional programming mechanisms, hence it is perhaps more likely that MT developers will introduce flaws into their specifications. These issues are more pronounced for declarative MT languages — in the case of QVT-R the issue of correct and appropriate semantics for the language is still an area of active research (Lano et al., 2019). Transformation processing can also be more conceptually difficult than for conventional applications, because of the complexity of the data being processed (models) and because of mutual dependencies between the processing of different model elements. ATL is the most mature of the MT languages, but still exhibits confusing language features, such as the distinctions between helper attributes and helper operations. For programming languages there are usually many publications, mature IDEs and other guidance and tool support, which can help to prevent flaws. MT languages lack such support and tooling (Burgueno et al., 2019). In some cases, poor tool support for MT languages can result in additional complexity in MT specifications because of the need to 'work around' tool limitations (Westfechtel, 2018a; Westfechtel and Buchmann, 2019).

There has been limited study of technical debt in functional or other declarative programming languages (Adams, 2017). The argument for functional languages is that TD can be removed by refactorings during development, because of the more powerful refactorings which functional languages support compared to traditional programming languages. This argument could also be applicable to the purely functional subset of the surveyed MT languages, however such a subset has considerable limitations: target elements cannot be created because this would result in a side-effect, and moreover none of the languages support functions as data, or higher-order functions.

Regarding the relative frequency of different kinds of flaws, (He et al., 2016) found that duplicate code was the most prevalent flaw in their 3GL cases, with cyclomatic complexity and too many methods also being relatively high. The survey of TD in Apache projects by Digkas et al. (2017) found that DC and CC were in the top 10 flaws and were widespread amongst the projects. DC and ENO are also in the top 5 most common flaws for model transformations, whilst CC is an issue only for ETL and UML-RSDS. In general, excessive coupling between rules/operations is apparently a more significant problem in MT than in 3GLs, this may be because of the additional forms of dependency in MT languages, especially in mechanisms for target-object lookup, the highly interdependent nature of data being processed, and because of the lack of well-defined architectures – particularly client–supplier architectures – for MT specifications (Kusel et al., 2015).

In summary for **RQ4**, it appears that transformations have a higher level of technical debt than programmes in traditional programming languages. They are also more prone to coupling/dependency flaws, but less widely affected by high cyclomatic complexity. However, duplicated code appears to be a flaw with similar frequency and distribution in transformations and programmes.

7. Technical debt and model transformation evolution

In this section we examine the relevance of our TD indicators as predictors of the change-proneness of elements (rules and operations) of a model transformation, and we also investigate how the flaw density of transformation cases changes over transformation evolution steps.

There are relatively few published transformation cases with an available change history. We identified the following 9 evolution cases:

- **ATL**: six versions of a UML to entity-relationship model refinement in Wimmer et al. (2012).
- **ETL**: first and second versions of an OCL to imperative QVT transformation. This is an industrial case.
- **QVT-R**: Original and rewritten versions of an expression normaliser and original and rewritten versions of the UML to RDBMS specification in QVT-R. Additionally, we consider three versions of the *ecore2sql* case of Westfechtel (2018a).
- **UML-RSDS**: multiple versions of the *uml2Ca* transformation, and single evolution steps for three other transformations: *cra*, *uml2Cb*, *uml2py*. These are large-scale cases representative of substantial applications of UML-RSDS.

7.1. ATL

Six versions of a UML to ER transformation are described in Wimmer et al. (2012). There are two evolution paths: the first is from version 1 to 2, to 3 and then to 4. The second is from 1 to 2 to 5 and then to 6. Thus there are 5 distinct evolution steps. (See Tables 28 and 29)

The first version has a single very large rule ($c = 131$). Version 2 refactors this to remove cloned expressions into new helper operations. This reduces the flaws from 3 to 2. Version 3 refactors the main rule by introducing three new matched rules. This removes the CBR_1 flaw, but increases size and *UEX*. Version 4 introduces an inheritance hierarchy for the additional matched rules to factor out parts of the new rule functionality. In versions 5 and 6, lazy rules are used instead of matched rules. This functional style reduces transformation size and *UEX* in version 6 compared to version 4.

Over the 5 evolution steps, 1 flawed element was changed in one step (the main rule in the step from 1 to 2). The flawed CBR_2 helper was not changed in any step.

7.2. ETL

Table 30 shows the indicator values for the two versions of the *ocl2qvtp* transformation from github.com/phillipus85/ ETL-Metrics/.

The transformation has been substantially rewritten and reorganised from the initial to the second version, however many of the same flaws (excessive number of operations and calls, cases of *EFO* and *ERS*, etc.) remain unchanged. The main improvement has been a reduction in code clones due to rationalisation and refactoring. The style of the specification has been made more functional and deterministic (there were 3 non-lazy rules in the

Table 28
TD indicators of ATL case versions of Wimmer et al. (2012).

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*CC*	*CBR*	*DC*	*UEX*
UML2ER (Version 1)	39 (35, 4)	1	1	1	0	0	2(1)	1	0
UML2ER (Version 2)	39 (29, 10)	1	3	0	0	0	5(1)	0	0
UML2ER (Version 3)	60 (50, 10)	4	3	0	0	0	5(1)	0	6
UML2ER (Version 4)	46 (36, 10)	6	3	0	0	0	8(1)	0	6
UML2ER (Version 5)	51 (41, 10)	4	3	0	0	0	8(1)	0	0
UML2ER (Version 6)	40 (30, 10)	6	3	0	0	0	8(1)	0	0

Table 29
Summary for ATL case versions of Wimmer et al. (2012).

Transformation	*Category*	*Style*	*LOC*	$c(\tau)$	*% in rules*	*# flaws*	*flaws/LOC*
UML2ER (1)	Refinement	L	39	149	90%	3	0.077
UML2ER (2)	Refinement	L	39	136	74%	2	0.051
UML2ER (3)	Refinement	L	60	181	83%	1	0.017
UML2ER (4)	Refinement	L	46	135	78%	1	0.022
UML2ER (5)	Refinement	F	51	145	80%	2	0.039
UML2ER (6)	Refinement	F	40	112	75%	1	0.025

Table 30
Technical debt indicator evolution for ETL.

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EFO*	*EFI*	*EPL*	*CC*	*CBR*	*DC*	*UEX*
ocl2qvtp	469 (264, 205)	6	11	2	1	1	0	2	0	27(0)	10	3
ocl2qvtp_v2	460 (208, 252)	6	18	2	1	2	0	1	0	39(0)	3	0

Table 31
Evolution summary for ETL.

Transformation	*Category*	*Style*	*LOC*	$c(\tau)$	*% in rules*	*# flaws*	*flaws/LOC*
*ocl2qvtp**	Refinement	I	469	1477	56%	18	0.038
*ocl2qvtp_v2**	Refinement	I	460	1424	45%	11	0.024

original version, only 1 in the revised). Overall the flaw count and flaw density have been reduced (Table 31).

Changes have been made predominately in the elements (rules or operations) which contained flaws: 11 of 17 elements contained flaws, and 9 of those 11 elements were modified (82%), including 2 cases of rule splitting, and 3 cases of deletion (major changes). There was one refactoring and two minor rewrites to the flawed elements, in 8 of these changed elements the overall number of flaws in the revised elements were reduced, and in one case there was no change. Only one element with no flaws was revised (17% of such elements). Thus the presence of flaws in an element is a good predictor of its change-proneness in this case.

7.3. QVT-R

Four elements of the DNF transformation with flaws are modified or removed in DNF_bbox (50% of elements with flaws). The revised version has lower total flaws (Table 32).

The classic UML to RDBMS transformation in its original form has 3 top relations, 4 non-top relations and one function. Three of the non-top relations are in a calling cycle, and there is one overlong relation also with EPL (Table 32). All these 4 elements with flaws are rewritten in the revised version, which has no flaws.

The *ecore2sql* case of Westfechtel (2018a) was successively simplified and refactored from version 1 to version 3 to remove flaws. The number of flaws was reduced in each step, together with the flaw density. Call graph complexity was consistently reduced, eliminating several EFI and EFO flaws. ERS and EPL flaws were also removed by rule simplification. However some flaws persisted across all three versions, such as self-recursively defined query functions. 11 of 17 elements with flaws are modified in the step from version 1 to version 2, and 10 elements from 16 with flaws are modified in the step from version 2 to version 3.

Overall, 64% of flawed elements in one QVT-R specification are modified in the step to the next version and there is a reduction of flaws in each transformation (Table 33).

7.4. UML-RSDS

There is a documented history of versions of the code generator transformations *uml2Ca*, *uml2Cb* and *uml2py* over 8 versions for the C generators, and 2 versions for *uml2py* and *cra*. For the code generators, the histories also include detailed effort estimates, enabling us to assign approximate costs to different kinds of modification action to remove flaws (Table 50). Table 49 shows the change in TD indicators over time for *uml2Ca*. Summarised data for all the UML-RSDS cases is in Table 34.

The growth of code size and the change in flaw density for *uml2Ca* are shown on Fig. 15. While there has been a general upward trend in the size of *uml2Ca* over time, and in the number of flaws, the flaw density does not show a definite trend.

Overall, 34% of elements with flaws in version n of *uml2Ca* are modified in version $n + 1$. This varies in particular versions from 0% to 72%. Considering the entire development history,

Table 32
Technical debt indicator evolution for QVT-R.

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EFO*	*EFI*	*EPL*	*CBR*	*DC*	*UEX*
DNF	396 (396,0)	9	0	4	0	0	0	4	10(4)	3	6
DNF_bbox	263 (263,0)	5	0	4	0	0	0	4	3(0)	3	6
UmlToRdbms	238 (226,12)	7	1	1	0	0	0	1	10(3)	0	0
uml2rdb2 (bx)	92 (92,0)	7	0	0	0	0	0	0	7(0)	0	7
ecore2sql1	1120 (784,336)	12	51	8	0	8	8	6	118(3)	5	0
ecore2sql2	956 (799,157)	12	26	6	0	6	6	6	88(3)	3	0
ecore2sql3	960 (769,191)	13	25	7	0	6	5	3	83(3)	3	0

Table 33
Results summary for QVT-R evolution cases.

Transformation	*Category*	*Style*	*LOC*	$c(\tau)$	*% in rules*	*# flaws*	*flaws/LOC*
DNF	Refactoring	L	396	665	100%	16	0.04
DNF_bbox	Refactoring	L	263	470	100%	11	0.042
UmlToRdbms	Refinement	F	238	314	95%	6	0.025
uml2rdb2 (bx)	Bidirectional	L	92	292	100%	0	0
ecore2sql1	Bidirectional	F	1120	1813	70%	42	0.038
ecore2sql2	Bidirectional	F	956	1428	84%	34	0.036
ecore2sql3	Bidirectional	F	960	1453	80%	31	0.032

Table 34
Results summary for UML-RSDS evolution cases.

Transformation	*Category*	*Style*	*LOC*	$c(\tau)$	*% in rules*	*# flaws*	*flaws/LOC*
uml2Ca V0.1/0.2	Code generation	F	126	116	0%	5	0.039
uml2Ca V0.3	Code generation	L	569	842	87%	8	0.014
uml2Ca V0.4	Code generation	L	616	896	88%	9	0.015
uml2Ca V0.5	Code generation	L	972	1593	65%	17	0.017
uml2Ca V0.6	Code generation	L/F	1389	2108	50%	21	0.015
uml2Ca V0.7	Code generation	L/F	1400	1560	58%	18	0.013
uml2Ca V0.8	Code generation	L/F	1408	1888	61%	21	0.015
uml2Ca V0.9	Code generation	L	874	1272	69%	22	0.025
uml2Cb V0.9	Code generation	F	1576	5621	16%	119	0.075
uml2Cb V1.0	Code generation	F	3618	8743	12%	82	0.023
cra (2016)	Refactoring	H	490	1360	32%	12	0.024
cra (2017)	Refactoring	H	844	1635	45%	12	0.014
uml2py V0.4	Code generation	F	605	3157	6%	50	0.083
uml2py V0.5	Code generation	F	1628	4846	2%	69	0.042

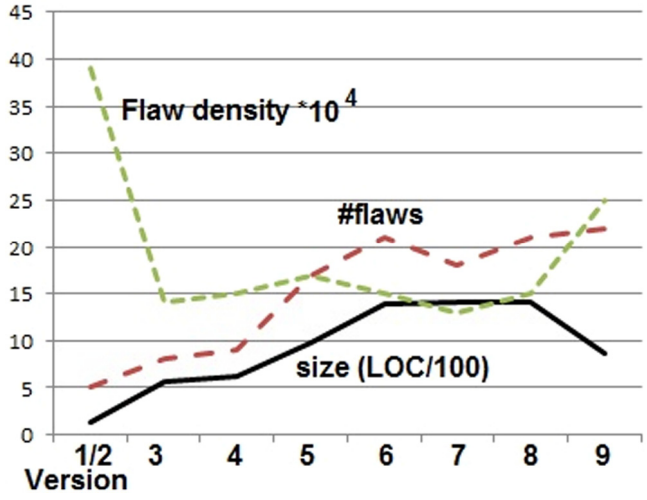

Fig. 15. Evolution of size, flaws and flaw density in *uml2Ca*.

39% of elements with flaws in some version n are modified in some later version m, $m > n$. This shows that in the evolution of *uml2Ca* existing flaws were usually not corrected in later versions, instead at most steps the evolution consisted mainly of the introduction of new functionality, with small amounts of rewriting and refactoring of existing elements. Quality flaws were tolerated if the elements were functionally correct. The step from version 0.6 to 0.7 involved major refactoring, which also reduced flaws. But only in the step from Version 0.8 to 0.9 was a specific effort made to remove quality flaws. In this step 58% of elements with flaws were revised. In particular, two overlong rules from *program2C* were simplified to reduce their size, the *printcode* transformation was simplified, and one EHS flaw removed. Six cases of cyclic dependencies in the operations of *printcode* were removed. The cyclic dependency flaws were considered to be causing high maintenance costs due to the complex inter-relations between operations: changes to one operation would require analysis of/changes to its mutual dependents. The refactoring led to higher numbers of code clones, but these were considered a less serious flaw. The time cost of this evolution step was 17 h, of which 8 h were spent on refactoring.

The situation with *uml2py* is similar. Development of *uml2py* was accelerated by reusing specification structure and operations from *uml2C*, however this reuse also replicated flawed structures (such as mutually-recursive operations for mapping UML expressions to code). Thus the level of flaws in *uml2py* is similar to that of *uml2C*. The step from *uml2py* Version 0.4 to Version 0.5 involved a substantial extension of functionality (eg., Python code generation for inheritance, object management and function caching was introduced) but also aimed to correct high-priority flaws ($CBR_2 > 0$) and 4 cases of these were removed from *stat2py*, together with 7 overlong operations from *exp2py*. However overall the number of flaws increased but with decreased flaw density. 50% of elements with flaws were modified. The time cost of the evolution step was 21 h, of which 8 h was spent on refactoring.

The evolution of *cra* from the 2016 version to the 2017 version also reduced flaw density. The functionality was rationalised with

Table 35
Technical debt indicators for *uml2Cb* Version 1.0.

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EFO*	*EFI*	*CC*	*CBR*	*DC*
uml2Cb Version 1.0	8743 (1049, 7694)	43	147	0	20	4	5	7	255(30)	8
(i) *exp2C*	5860 (0, 5860)	0	75	0	20	4	4	7	139(7)	4
(ii) *stat2C*	667 (138, 529)	8	26	0	0	0	0	0	49(10)	1
(ii) *printcode*	2216 (911, 1305)	35	46	0	0	0	1	0	67(13)	3

Table 36
Development costs for UML-RSDS evolution steps.

Transformation	*Total cost (hours)*	*Refactoring cost (hours)*	*Size (c)*	*#flaws*	*#flaws removed*
uml2Ca V0.8 to V0.9	17	8 (47%)	1888	21	10
uml2py V0.4 to V0.5	21	8 (38%)	3157	50	11
uml2Cb V0.9 to V1.0	70	14 (20%)	5621	119	37

a reduced number of operations. Two flaws of excessive *CC* were removed, and CBR_1 was reduced. 33% of elements with flaws were modified. Overall the number of flaws was unchanged.

For *uml2Cb*, between Version 0.9 and Version 1.0 specific flaw-reduction work was carried out, to split excessively-large operations and to subdivide the excessively-large *exp2C* transformation. There was also significant expansion of functionality. Table 35 shows the flaw data for this version. The specific actions that were taken were:

(1) Internal separation of *exp2C* into two transformations, *exp2C* and *stat2C*.
(2) Factoring of duplicated code into new auxiliary operations. This also reduced the number of cases of excessive *CC*, however it resulted in an increase in CBR_1.
(3) Operation splitting of the two largest operations into four sub-operations each.
(4) Removal of one EFO case by operation splitting.

Out of 78 elements with flaws, 23 were modified, a rate of 29%. Overall the number of flaws were reduced by 37, and the flaw density reduced to 0.023, below the average for UML-RSDS. The time cost for the evolution step was 70 h, of which 14 h were spent on refactoring.

In general, development costs of evolution steps appear to increase with the initial transformation size and number of flaws (Table 36). In each case significant effort was needed to correct flaws, however the presence of flaws also caused increased costs during evolution steps: overlarge transformations and operations increased the effort of locating bugs or making changes to the specification, as did complex dependencies and excessive numbers of local variables and logical conditions. Duplicated expressions led to the duplication of bug fixing and enhancement work. Although the particular style of expression and statement mapping adopted at the core of *exp2C* and *stat2C* can be regarded as a pattern which provides a systematic organisation of the mapping (the Inverse Recursive Descent pattern (Lano et al., 2019)), the style results in high numbers of CBR_2 flaws. Map Objects before Links is an alternative structure, but this also has disadvantages. Thus we have not attempted to alter this aspect of the code generators.

The typical costs of the individual refactoring actions are given in Table 50. Eg., splitting the operation *mapAttributeExpression* of *uml2Cb* into four new supplier operations took 15 min to perform the text editing and a further 15 min for testing.

Table 37 shows the precision and recall figures for *"presence of flaws in an element"* as a predictor of *"element is modified in the next evolution step"*, aggregated for the cases in each of the languages. *precision* is $\frac{\#(modified, flawed)}{\#(total\ modified)}$ and *recall* is $\frac{\#(modified, flawed)}{\#(total\ flawed)}$. The *f-measure* is also shown, this is $2 * \frac{precision * recall}{precision + recall}$.

Table 37
Precision and recall for evolution cases.

	ATL	*ETL*	*QVT-R*	*UML-RSDS*
Precision	0.2	0.9	0.47	0.38
Recall	0.5	0.82	0.64	0.27
F-measure	0.29	0.86	0.55	0.32

This shows that our TD indicators are a good predictor of change-proneness in some cases (ETL and QVT-R) but are only weakly related to changes in the UML-RSDS cases considered, which were evolving in a way that mainly added new functionality, retaining existing specification elements unchanged. However for UML-RSDS evolution steps the number of flaws at the start of the step is highly correlated with the cost of the step (Table 36). The changes to the ATL case did improve the overall quality of the case, but this was not reflected in the flaws of individual elements except in one step.

Research into the evolution of code technical debt has shown that this typically increases over time for an application as it is modified and extended (Digkas et al., 2016, 2017; Zazworka et al., 2014), however the normalised TD per LOC may decrease as an application evolves (Digkas et al., 2017) or may have no clear trend (Zazworka et al., 2014). Other work on the evolution of quality flaws has shown a tendency for relatively few code smells to be removed during an application's lifetime, even when developers are aware of the flaws (Peters and Zaidman, 2012; Tufano et al., 2015). We observed this tendency also in the UML-RSDS transformation cases.

8. Threats to validity

Threats to validity include bias due to the internal construction of the experiment, inability to generalise the results, inappropriate constructs (i.e., the TD concept being used) and inappropriate measures.

8.1. Threats to internal validity

Instrumental bias. This concerns the consistency of measures over the course of the experiment. To ensure consistency, all measurement was carried out in the same manner by a single individual (the second author) on all cases. In addition, the measurement process was repeated for all cases in order to double-check the scores.

Selection bias. Cases were selected independently of their TD or design quality (which were unknown at the point of selection). Where possible, we selected cases written by experts in the languages. At least 11 of 25 ETL cases were written by experts in the language, similarly for 22 of 25 QVT-R cases, and all the

Table 38
Adjusted overall size and quality flaw results.

Language	LOC	*c*	*c*/LOC	Flaws	Flaws/ LOC	Flaws/ *c*
ATL	11,398	21,185	1.86	222	0.0195	0.0105
ETL	6722	20,395	3.03	172	0.026	0.008
QVT-R	10,793	19,435	1.8	311	0.029	0.016
UML-RSDS	6437	16,498	2.6	173	0.027	0.01
Overall	35,350	77,513	2.19	878	0.025	0.011

Table 39
Percentages of transformations/elements over thresholds.

Threshold	*ATL*	*ETL*	*QVT-R*	*UML-RSDS*
ETS	18.4%	10.5%	***24%***	8.7%
ENR	***31.6%***	5.3%	***36%***	10.5%
ENO	23.7%	15.8%	20%	***35%***
ERS	6.4%	4.2%	***21.4%***	4.3%
EHS	0.7%	8%	1%	5.7%
EPL	0.8%	3.7%	***17.5%***	0.1%
EFO	2.7%	3.2%	7%	1.3%
EFI	2.7%	1.4%	4.8%	0%
CC	0.65%	3.1%	0.5%	7.4%

UML-RSDS cases. For ATL the proportion is not known, however the cases are published as examples of ATL in the ATL zoo and mainly originate either from the INRIA ATL team or from the industrial collaborator, Sodius. We also aimed to obtain similar size distributions for the cases in different languages, and this was achieved (Fig. 1).

The average size of the datasets are different. However, if we adjust the set of UML-RSDS cases by removing the 2 largest UML-RSDS cases, we obtain average *c*-sizes of 847.4 for ATL, 777.4 for QVT-R, 816 for ETL and 717.3 for UML-RSDS, which are within 15% of each other. The adjusted overall results for these sets of cases are shown in Table 38. These are similar to the unadjusted results, although there is a relative improvement in the UML-RSDS flaw density figures.

8.2. Threats to external validity

Generalisation to different samples. In this study we have only considered manually-constructed MT specifications, in four specific languages. Our study does not generalise to automatically-generated transformations, which seem to have different characteristics, dependent on the generation process (Kapova et al., 2010). The samples we have taken are otherwise typical of the transformations available in public repositories for the languages, and include both industrial and academic cases. Some of the cases are well-known and are widely-used as examples of transformation specification (eg., the UML to RDBMS transformations, and state machine flattening/unflattening). Thus our results should be relevant to other samples of transformations in the languages. The noted tendencies in flaw density differences between the languages were already present in our initial study (Lano et al., 2018a) with 8 ATL cases, 13 ETL, 12 QVT-R and 10 UML-RSDS. An intermediate study with 20 ATL cases, 15 ETL, 14 QVT-R and 16 UML-RSDS also showed similar results. Both studies also ranked ATL below QVT-R and ETL in terms of flaw density wrt LOC, and QVT-R below UML-RSDS. The 6 most-frequent flaws were the same in Lano et al. (2018a) as in this paper, with CBR and then DC as the top two flaws. This consistency also suggests that the results can be generalised.

In terms of generalisation to different languages, we have not considered imperative transformation languages such as Kermeta, EGL or QVT-O in this paper. Initial investigation into design flaws in these languages is described in Kolahdouz-Rahimi et al. (2020a). Declarative graph-transformation languages such as Triple Graph Grammars (Hermann et al., 2014) and hybrid languages such as GrGen (Jakumeit et al., 2010) have similarities to QVT-R and ATL/ETL, and it is likely that similar approaches to identifying TD could be applied to these languages.

8.3. Threats to construct validity

Inexact characterisation of constructs. We have focussed on TD due to quality flaws in the transformation specifications. This is an incomplete characterisation of TD since other issues may influence TD, such as lack of documentation or incomplete testing. However, precise estimation of such additional factors is difficult to obtain in practice, and our approach to TD identification is widely adopted in TD research and tools (He et al., 2016; Kosti et al., 2017; Marinescu, 2012; Zazworka et al., 2014). The close correlations between size in LOC and in *c* (Table 18) suggests that the *c* measure is valid as an alternative size measure.

The thresholds we have chosen for indicators are also open to question. We can analyse the suitability of these thresholds by considering what proportion of transformations or elements exceed the thresholds (Table 39). This shows that for ETS the threshold of 500 LOC may be too low. Increasing the threshold for all languages to 1000 LOC results in 4.4% of transformations being over the threshold. Likewise for ENR/ENO: increasing the ENR/ENO thresholds to 20 results in 6.3% of transformations being over the ENR threshold, and 10.1% over the ENO threshold. For ERS and EHS the languages each have approximately 1%–7% of rules or operations over the threshold, except for the case of rules in QVT-R. We consider that this high rate of ERS is a feature of the language, due to the lack of facilities in QVT-R for decomposing rules, and the limited capabilities of QVT-R operations. Similarly for EPL the high percentage for QVT-R is a characteristic of the language. Overall we can conclude that the thresholds for ERS, EHS, CC, EFO and EFI seem reasonable in general, in distinguishing around 1%–8% of elements as flawed, however there is evidence that the thresholds for ETS and ENR/ENO are too low, and that some thresholds may need to be varied for different languages. In future work we will survey MT developers to gain input on alternative flaw thresholds.

We do not claim that our indicators cover all kinds of flaw that can indicate TD, and further indicators could be adopted, such as the presence of OCL specification flaws (Correa and Werner, 2007) and language-specific bad smells (Bonet et al., 2018). We discuss a more extensive quality model and measures in Section 10.

Redundancy of constructs. We can examine if some of the proposed measures are redundant or duplicate other measures, by computing the correlations of pairs of different measures, for each set of cases.

We found only a few pairs of measures with high correlations across several languages (Table 40). Although in a few cases there are statistically-significant correlations for individual languages (underlined correlations), there is no significant correlation between pairs of measures when cases across all languages are considered. Therefore we can conclude that none of the proposed measures are redundant wrt other measures.

Another form of redundancy is if the observed flaw densities can be explained by an individual measure or simple combination of them. Table 18 showed low correlations of transformation size with flaw density. We tested several other functions of one, two or three measures for correlation with flaw density (fd). Table 41 shows the correlations for average element size $\frac{ETS}{ENR+ENO}$, $CBR1$, DC, $ENO + ENR$ and average element coupling $\frac{CBR1}{ENR+ENO}$ against flaw density. These were the best overall combinations. Again,

Table 40
Correlations between measures.

Correlation	*ATL*	*ETL*	*QVT-R*	*UML-RSDS*
EFO/EFI	0.63	0.39	0.84	0
ENR/DC	0.59	−0.13	0.74	0.63
ENR/CBR1	0.58	0.26	0.69	0.53
ENO/CBR1	0.9	0.49	0.72	0.98
CBR1/DC	0.68	0.08	0.76	0.79
ETS/ENR	0.69	0.11	0.74	0.44
ETS/ENO	0.78	0.68	0.6	0.97
ETS/DC	0.84	0.47	0.79	0.74
ETS/CBR1	0.83	0.27	0.96	0.95

Table 41
Other measures influencing flaw density.

	average element size/fd	CBR1/ fd	DC/ fd	ENO + ENR/ fd	average element coupling/fd
ATL	0.11	0.12	−0.05	−0.14	0.39
ETL	−0.38	0.29	0.08	−0.34	0.86
QVT-R	0.32	0.32	0.55	0.22	0.47
UML-RSDS	0.29	0.63	0.51	0.53	0.36

the correlations are low, suggesting that no simple combination of measures are sufficient to explain observed flaw density.

Another factor that could be considered to influence flaw density is metamodel size (the number of classes + the number of features in the metamodel). However we also found low correlations of this factor with flaw density (ETL: 0.04; ATL: −0.24; QVT-R: 0.21). The highest magnitude of correlation is for UML-RSDS, where the correlation of metamodel size and flaw density is 0.51.

Metamodel size is also poorly correlated to the number of flaws: (ATL: 0.30; ETL: −0.14; QVT-R: 0.78; UML-RSDS: 0.76).

8.4. Threats to content validity

Relevance. The selected cases have been selected from public repositories of language cases, and mainly originate from experts in the respective languages. The cases in the ATL zoo have been used in several other studies of ATL transformations, and the ETL repositories used have also been the subject of the study of Bonet et al. (2018). The cases from the Epsilon repository are published by the ETL originators as examples of how to use ETL, likewise for the Modelmorf and Medini QVT-R examples and the UML-RSDS zoo examples. The ETL cases TTC17Live and StateElimination ETL cases were overall winners of their parts of the 2017 TTC competition (www.transformation-tool-contest.eu/2017).

Thus the sets of cases originating with the language teams can be considered as demonstrating the intended use of the languages, and these examples have an influence on how general users apply these languages. The fact that we identified systematic flaws of certain kinds in these 'ideal' examples perhaps shows that there are inherent issues with the languages.

Representativeness. For each language, we have endeavoured to obtain a wide range of transformation examples from the available published cases, spanning a wide range of sizes, and across the range of all available categories and styles of transformation.

The cases we obtained were predominately academic. There are 11 industrial cases available in the surveyed datasets, Table 42 shows the analysis results of these cases for each language, derived from Tables 51 and 52. The flaw density values are somewhat higher for these cases compared to the results in Table 15, for ATL, QVT-R and ETL, reflecting that these cases address more challenging problems than for average cases in the languages, but for UML-RSDS the density figures are lower for these cases, perhaps because the development process involved close collaboration with the customer and hence there was an emphasis on improving the clarity of the specification. Refactoring was also used during the development. Thus there is no evidence that the industrial cases are significantly different from academic cases in terms of their technical debt.[2]

Table 42
Flaw data for industrial cases.

Language	LOC	*c*	flaws	flaws/LOC	flaws/*c*
ATL	392	1015	12	0.031	0.012
ETL	2908	7225	77	0.026	0.011
QVT-R	2038	5415	63	0.031	0.012
UML-RSDS	846	2065	21	0.025	0.01

8.5. Threats to conclusion validity

We used the Shapiro–Wilk and Anderson–Darling tests to test for normality of the flaw density distributions. These are considered the most effective tests for this purpose (Razali and Wah, 2011). To test for differences in the distributions we used the Welch variant of the t-test and the Kolmogorov–Smirnov (K–S) test. The t-test compares the means of the distributions, but assumes normality in the compared distributions, and since this assumption is unsupported here, we also applied the K–S test, which is independent of normality assumptions. The K–S test takes account of differences both in the mean and the shape of the compared distributions.

9. Related work

One of the first works to consider metrics for MT was (Kapova et al., 2010). They define measures for the size and complexity of QVT-R transformations, including lines of code, number of relations (corresponding to number of rules), and specific measures for the size and inter-relationship of QVT-R rules. Their analysis is specific to QVT-R and does not consider clone detection or detailed analysis of the rule dependency graph. They evaluated one large auto-generated QVT-R transformation and three moderate/small manually-constructed transformations. In contrast to Kapova et al. (2010) we directly count the number of calling inter-dependencies between rules/operations, instead of counting when/where predicates (for QVT-R). in van Amstel et al. (2011a), measures are computed for ATL, QVT-R and QVT-O transformations for versions of two transformations in each language. Since the transformations are small-scale, no quality flaws are detected from the measures, although these help to illustrate differences in the structure of the three versions.

In van Amstel and van den Brand (2011c), seven ATL transformations are evaluated by metrics and by expert analysis, in order to identify correlations between metric values and expert evaluation of quality characteristics. Wimmer et al. (2012) use quality measures to evaluate the effect of MT refactorings. They adopt ERS, DC and EFO as quality criteria for ATL transformations, and measure the number of rules and helpers in a transformation, as well as its size in LOC. They propose an additional measure, maximum OCL expression length (MEL), of a transformation. They consider that OCL expressions with complexity greater than 10 are a potential quality flaw. This measure could also be incorporated into our framework and tools.

[2] One interesting difference was that self-declared technical debt was common in the industrial cases (7 of the 11 cases) but infrequent in academic cases.

Detection of transformation bad smells is carried out for ETL by Bonet et al. (2018). They relate ETL language construct measures such as the number of procedural statements and number of variables per rule, to the frequency of occurrence of bad smells such as duplicated complex expressions. They found that the number of variables per rule (i.e., EPL) and the number of calls to operations per rule both had significant effects on the occurrence of bad smells, as did the overuse of procedural constructs.

A detailed quality model for QVT-O is provided by Gerpheide et al. (2016). They include measures such as size in LOC and complexity (from Kolahdouz-Rahimi et al. (2014)), fan-out, EPL, DC, which correspond to the quality flaw measures we have adopted. In addition they consider aspects such as confluence (related to UEX), the presence of dead code and the absence of expected patterns. Support for QVT-O maintenance using visualisation of control and data dependencies is described in Rentschler et al. (2013). They found that understanding of such dependencies was an important factor in enabling efficient maintenance of QVT-O transformations.

Clone detection in transformations is considered by Struber et al. (2017), and they evaluate alternative tools for clone detection in graph transformations. The paper (He et al., 2016) analyses code-level technical debt in GMF/EMF-based MDE projects, and concludes that TD indicators are higher in automatically-generated code than in manually-written code, in these projects. The flaws may be due to flawed models or due to the code generator transformations themselves introducing flaws. This is an additional type of transformation flaw which is important to consider and reduce.

There have been several studies comparing different MT languages. The annual Transformation Tool Contest compares different languages on a variety of case studies. Ten of our cases were also TTC case solutions. in Kolahdouz-Rahimi et al. (2014) we compared UML-RSDS, QVT-R and ATL on a refactoring case using several quality measures. It was found that the call-graph complexity (CBR) of the ATL and QVT-R solutions were significantly higher than for the UML-RSDS solution.

An initial version of the present paper was published as Lano et al. (2018a) on a smaller set of cases (43 transformation systems instead of 100). The general conclusions and comparison results for the different languages were similar between the two studies. However in this paper we have selected a wider set of cases in order to obtain a more representative dataset, with similar size distributions for the case sets in each different language. We have provided rigorous statistical analysis of the results, and expanded our analysis to consider transformation evolution and TD reduction strategies. In this study we identified that ETL had a flaw density below QVT-R and UML-RSDS, in contrast to the findings of Lano et al. (2018a). This difference may be due to a more accurate estimation of flaw levels in cases, and a more representative set of cases.

The issue of MT modularisation is considered for ATL in Fleck et al. (2017). They aim to minimise element dependencies between modules, and include rule data-dependencies in addition to the calling dependencies considered in this paper (a rule R is considered to data-depend on rule P if R reads data written by P). They target the creation of separate modules which can be combined by ATL superposition, which is a form of module import/inheritance rather than a client–supplier relationship. Nonetheless the principles of this modularisation process could be useful in general for reorganising transformation specifications to reduce maintenance costs.

Different approaches for QVT-R specification are considered in Westfechtel (2018a). In particular three versions of a large bx transformation, *ecore2sql*, are defined using different styles, including recursive descent with non-top relations or a more logical style based primarily on top relations. The recursive descent version has fewer flaws due to reduced relation dependencies. in Westfechtel and Buchmann (2019) it is suggested that the complexity of bx specification in QVT-R can lead to poor quality specifications, and that it is preferable to use mutually-inverse unidirectional transformations instead. In our analysis we also found that bx specifications tended to have high quality flaw density.

A survey of issues with the usage of MT languages was reported in Burgueno et al. (2019). A recurrent view was that MT languages suffer from lack of maturity and effective tool support. In particular, lack of scalability was a significant issue: "... it was easy to do small examples in a MT language, but medium examples were unpleasant, and large examples awful — and nearly always better done in a GPL" (quoted survey response from Burgueno et al. (2019)). This concurs with our findings that flaw levels in QVT-R and UML-RSDS cases were particularly high for large cases. On the other hand, ETL transformations tend to increasingly resemble GPL programmes in large cases.

There has been substantial research on technical debt and related quality concepts such as code smells in programming languages (Lacerda et al., 2020). Whilst we have focussed on issues in MT languages, there are common aspects to this research, in particular many of the widely-used code smells identified in the programming domain are relevant to MT (eg., smells of excessive element size, excessive numbers of variables, duplicated code and excessive coupling). We have used these smells as indicators of MT technical debt.

10. Future work

We intend to extend the model transformation quality model presented in Section 2.1 with quality requirements and measures/indicators for other quality aspects, such as performance and style.

Table 43 lists possible additional requirements, measures and the relevant quality characteristics.

The extended quality model would use additional measures:

- *MEL* — Maximum expression length from Wimmer et al. (2012).
- *OBS* — a count of OCL bad smells from Correa and Werner (2007), Cabot and Teniente (2007).
- *UOP* — count of the number of unused operations + unused called/lazy/non-top rules per transformation.
- *UVA* — for each operation or rule, a count of the number of unused parameters or local variables in the element.
- *SDTD* — number of code comments which indicate incompleteness or unresolved issues in the specification, eg., "TODO", "FIX THIS" (Maldonado et al., 2017; Wehaibi et al., 2016).

In some cases it is difficult to automatically measure the satisfaction/violation of these quality requirements. Manual intervention would probably be needed in some cases, for example only certain MT patterns can be automatically recognised, and only a subset of SDTD comments can be automatically identified (Maldonado et al., 2017; Wehaibi et al., 2016). It is our aim to automate as many of these quality measure as possible, and to re-evaluate our datasets using this extended quality model.

In addition, the following research questions are important to address in future work:

FRQ1 : Do the TD indicator results correlate with expert judgement regarding TD in cases?

FRQ2 : Are the analysis results for industrial MT cases significantly different to those of academic cases?

Table 43
Extensions of the model transformation quality model.

Category	*Quality requirement*	*Measure/threshold*	*Quality characteristics* Letouzey (2016)
Semantic Complexity	Functionality should be complete	Rules exist for all cases of each source metamodel class	Maintainability, Usability
	There should not be high expression sizes in elements	$MEL \leq 10$ for each rule & operation	Testability, Changeability
Performance	Utilise caching of recursive/EFI operations where possible	*#uncached recursive or EFI query operations with discrete typed parameters* $= 0$	Efficiency
	Avoid inefficient OCL expressions eg., $\rightarrow select(..) \rightarrow first$ or $\rightarrow select(key = value)$ expressions	$OBS = 0$	Efficiency
Redundancy	There should be no unused operations/rules in the transformation	$UOP = 0$	Maintainability, Testability
	There should be no unused variables in elements	$UVA = 0$	Maintainability, Testability
Style	Design patterns should be correctly used	*#correct pattern uses* $-$ (*#incorrect uses* $+$ *#missing uses*)	Maintainability, Reusability
	Rule inheritance should match source metamodel inheritance	*#incorrect rule inheritances* $= 0$	Changeability, Maintainability, Reusability
	No self-declared technical debt	SDTD = 0	Reliability, Maintainability
	No use of depricated constructs	*#of uses of depricated constructs* $= 0$	Maintainability, Reusability

FRQ3 : Can a time estimate of TD remediation and non-remediation costs be produced for MT, which agrees with empirical data on such costs?

FRQ4 : Is it possible to identify the contribution of metamodel flaws and metamodel TD to the maintenance costs of model transformations operating on the metamodel?

FRQ5 : Is it possible to actively manage TD in MT?

FRQ6 : Is maintenance of MT more time-consuming than maintenance of programmes?

FRQ1 can be addressed by using a survey to solicit expert views on the TD level of selected cases, and analysing how these correspond to our adopted measures and results for the same cases.

FRQ2 can be addressed by a survey specifically directed at industrial cases, which we will aim to access via the development groups of the respective MT languages, and via our own industrial contacts in the MDE community.

FRQ3 would require a longitudinal study of transformation evolution cases in multiple languages, which may be difficult to obtain. As with FRQ2, the development groups of the different languages may be the most suitable source for such cases.

FRQ4 would also require a dataset with evolution and maintenance data, and tooling and analysis to identify metamodel flaws, eg., as in Bettini et al. (2019), Strittmatter et al. (2016). Some potential effects of metamodel flaws are:

- Concrete superclass CS of subclasses Sub_1, …, Sub_n: rules which should deal specifically with $x : CS$ instances must have conditions to exclude that x belongs to any Sub_i class. This increases rule size and CC.
- Excessive depth of inheritance hierarchies; excessive numbers of subclasses: excessive numbers of rules, with rule dependencies for each inheritance relationship.
- Multiple inheritance: potential consistency problems in rules resulting in conflicts between the multiple generalised rules.
- Duplicate features in sibling classes: DC in rules which process instances of these classes, which cannot be removed by refactoring common assignments into a generalised rule.

FRQ5 would require the development of tooling to support MT refactoring based on TD indicators. Possible refactorings and their effect are discussed in Appendix B. At present, our tools identify violations of quality requirements, for ATL, ETL, QVT-R and UML-RSDS (Fig. 2), but do not suggest or perform refactorings to remove such violations.

FRQ6 would involve analysing the effort of maintenance actions upon comparable transformations and programmes. The study of Hebig et al. (2018) performed this comparison for ATL, QVT-O and Xtend, and found higher effort was needed for maintenance actions in transformation language cases. However this was a study carried out using students as subjects, and different results may be obtained using experts as subjects.

Other interesting areas of research would be to examine if the use of MT design patterns reduces technical debt in transformations, and to compare the transformation synthesis strategies of different automated MT generation approaches (such as Goldschmidt and Wachsmuth (2011) and Lano and Fang (2020)) with regard to the quality of the generated transformations.

11. Conclusions

We have shown that quality flaw indicators for technical debt can be evaluated in different MT languages. In our main analysis we evaluated 158 transformations and 100 transformation systems in four transformation languages, and answered the research questions.

For **RQ1** we found significant quality flaws and hence potential technical debt across all the languages and the majority of transformations and cases.

For **RQ2** we found that coupling and duplicate code issues were clearly the most common flaws in MT cases, followed by issues of excessive transformation size and rule size.

For **RQ3** we identified significant differences between the languages in terms of the frequency and type of quality flaws: UML-RSDS cases are particularly affected by flaws due to excessive use of operations, whilst QVT-R cases are affected more by excessive numbers of local variables. The variability of flaw levels is high in the analysed ETL cases compared to the ATL cases, possibly due to the more diverse range of specification styles

supported by ETL. We found statistically significant differences between the flaw distribution of ETL cases and that of ATL cases. Whilst the declarative nature of QVT-R and UML-RSDS produces transformations with low numbers of flaws in small or medium-sized cases, larger cases in these languages become excessively complex and can result in high levels of flaws.

For **RQ4** we found that cases in all the MT languages are impacted by flaws due to complex dependencies between rules/operations, to a greater extent than for software written in programming languages. This may be a symptom of poor modularisation facilities in MT languages, and the wider range of dependency mechanisms in these languages.

The conclusions must be qualified by the necessarily limited nature of the analysis carried out here, which was restricted to publically-available transformation cases. It is possible that larger-scale commercial uses of transformations would exhibit different patterns of flaws.

The identification of quality flaws can help MT specifiers to improve their transformations and to prioritise refactoring or other quality improvement work on their transformations. More generally, by identifying the characteristics of quality flaws in transformations, we can enable developers to improve the quality of MT specifications.

CRediT authorship contribution statement

Shekoufeh Kolahdouz-Rahimi: Conceptualization, Investigation, Methodology, Writing - original draft, Writing - review & editing, Data curation. **Kevin Lano:** Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing - original draft, Writing - review & editing. **Mohammadreza Sharbaf:** Data curation, Software. **Meysam Karimi:** Data curation. **Hessa Alfraihi:** Data curation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Detailed analysis results

Tables 44–48, show the measures of quality flaws for individual transformations, for each ATL, ETL, QVT-R and UML-RSDS case. Underlined measures in Tables 44–48 identify where thresholds of the measures are exceeded, i.e., when specific flaws and TD indications occur in cases.

Table 49 shows changes in quality flaws for *uml2Ca* over different versions.

Additional industrial cases were also selected for ETL. Tables 51 and 52 give the data of all the industrial cases in the dataset.

Appendix B. Reducing and managing technical debt in MT

The technical debt indicators help to identify problem areas in particular transformations, such as rules which exceed the thresholds in terms of size, inter-dependencies, etc. Such flaws can be removed by refactoring of the affected rules.

B.1. Transformation refactoring

Because of the similarities between the MT languages identified in Section 4, some refactorings are applicable to all of these languages, whilst others are language-specific, such as replacing implicit calls by explicit calls for ETL.

In general, if a rule r exceeds the size threshold of *ERS* wrt c, $c(r) > 100$, the specifier could identify a subpart P of r which can be factored out as a separate called rule or operation $r1$. The complexity of $r1$ is $c(P)$, whilst $c(r)$ is reduced by $c(P) - x$ where x is the complexity of a call to $r1$. $c(P)$ should be significantly larger than x, and sufficiently so to ensure that $c(r) - c(P) + x < 100$, whilst $c(P) < 100$. The same refactoring applies to operations that exceed size thresholds (*EHS*). Similarly, rules or operations which are over the 50 LOC limit can be refactored by this rule/operation splitting refactoring.

The operations *mapAttributeExpression* and *mapRoleExpression* from *uml2Cb* version 0.9 are examples of this situation: both have c measures over 300 (the worst examples in this transformation). They are also the source of some of the largest cases of duplicated code in the transformation. In the step to Version 1.0 the operations were each refactored into four new subordinate operations to handle specific cases of attribute/role basic expressions. This refactoring is correct-by-construction, because an operation of the form

```
op(pars : types) : E
pre: P
post:
 ( C1 => E1->exists( e1 | P1 & result = e1 ) )
 & Cases
```

is rewritten to:

```
op1(pars : types) : E1
pre: P
post:
 E1->exists( e1 | P1 & result = e1 )

op(pars : types) : E
pre: P
post:
 ( C1 => result = op1(pars) ) & Cases
```

where $E1$ is a subclass of E. The new operations were then further rationalised to remove duplicated code.

Such a refactoring can reduce *DC* and *CC* measures, but may increase the measures *ENR* or *ENO* of number of rules/operations, and the number of call dependency flaws CBR_1, *EFO* and *EFI*.

To reduce the overall size of a large transformation, it can be factored into a sequential composition of smaller transformations. This can be performed in two ways, either an *external* sequential composition with one transformation producing an explicit output model which can be read by its successor(s) (the Transformation Chain MT pattern (Lano et al., 2018b)), or an *internal* sequential composition with the intermediate model held internally within the transformation. An example of decomposition into internally sequenced subtransformations is the introduction of 'phases' within a transformation (Cuadrado and Molina, 2009). Either form of decomposition can reduce the *UEX* metric and others for the individual subtransformations. The use of *lazy* rules in ATL and ETL instead of matched rules (non-top relations instead of top relations in QVT-R) can also reduce *UEX*.

The existence of mutual dependencies between (top-level) rules is an indicator that the Map Objects before Links pattern should be applied to avoid the dependency cycles. This pattern maps entity instances and their attributes in a first phase, and

Table 44
Technical debt indicator evaluations for ATL cases.

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EPL*	*EFO*	*EFI*	*CC*	*CBR*	*DC*	*UEX*
Families2Persons	39 (15, 24)	2	2	0	0	0	0	0	0	4(0)	0	1
Eliminate Inheritance	53 (31, 22)	3	2	0	0	0	0	0	0	3(2)	0	3
Public2Private	79 (76, 3)	2	1	0	0	0	0	0	0	2(0)	2	1
Replace Association by Attribute	84 (78, 6)	6	1	0	0	0	0	0	0	1(0)	1	15
Class2Relational	97 (94, 3)	6	1	0	0	0	0	0	0	6(0)	1	15
cdrat	105 (105, 0)	6	0	1	0	0	0	0	0	1(0)	1	10
EMF2KM3	118 (109, 9)	10	1	0	0	0	0	0	0	1(0)	0	36
Merging Partial Classes	185 (180,5)	10	2	0	0	0	0	0	0	4(0)	3	36
PartialRolesTotal	243 (238, 5)	9	1	1	0	0	0	0	3	2(0)	0	36
AssertionModification	273 (159, 114)	13	7	0	1	0	0	0	0	13(0)	4	66
SimpleClass2 SimpleRDBMS	302 (166, 136)	1	6	1	0	1	1	0	0	13(6)	0	0
Excel Extractor	311 (251,60)	13	5	0	0	0	0	1	0	15(1)	2	66
(i) *SpreadsheetML Simplified2XML*	263 (246,17)	12	1	0	0	0	0	1	0	10(0)	2	66
(ii) *XML2ExcelText*	48 (5,43)	1	4	0	0	0	0	0	0	5(1)	0	0
MDL2GMF	384 (172, 212)	7	26	0	0	0	0	1	0	43(0)	1	15
KM32Measure	392 (224, 168)	6	23	2	0	0	3	0	0	52(5)	0	6
Excel Injector	395 (231,164)	11	10	0	0	0	0	2	0	39(0)	3	55
KM3 to DOT	451 (251,200)	7	18	1	0	0	1	0	0	33(0)	4	21
Make to Ant	456 (330,126)	18	11	0	0	0	0	0	0	17(2)	3	41
(i) *XML2Make*	147 (73,74)	5	7	0	0	0	0	0	0	11(1)	0	10
(ii) *Make2Ant*	88 (88,0)	5	0	0	0	0	0	0	0	0(0)	1	10
(iii) *Ant2XML*	177 (164,13)	7	1	0	0	0	0	0	0	2(0)	2	21
(iv) *XML2Text*	44 (5,39)	1	3	0	0	0	0	0	0	4(1)	0	0
FM2BPMN	474 (357, 117)	12	18	1	0	1	6	5	0	79(0)	4	45
UML2MOF	585 (465, 120)	12	7	1	0	0	0	1	0	13(0)	3	66
MDL2UML21	645 (477, 168)	18	18	4	0	1	0	2	0	36(0)	3	153
Monitor2Semaphore	886 (786, 100)	24	10	2	0	2	1	0	0	44(0)	2	276
MOF to UML	935 (746, 189)	11	11	5	0	0	0	1	0	23(0)	7	55
MySQL to KM3	995 (571, 424)	20	28	1	0	0	1	0	1	62(4)	7	71
(i) *XML2XML*	101 (87, 14)	4	1	0	0	0	0	0	0	2(0)	2	6
(ii) *XML2MySQL*	281 (137,144)	5	10	0	0	0	0	0	0	19(2)	2	10
(iii) *MySQL2KM3*	613 (347,266)	11	17	1	0	0	1	0	1	41(2)	3	55
Maven to Ant	1307 (1139,168)	90	18	0	0	0	0	2	0	80(0)	7	1326
(i) *XML2Maven*	575 (472,103)	36	13	0	0	0	0	2	0	74(0)	3	630
(ii) *Maven2Ant*	360 (308,52)	30	4	0	0	0	0	0	0	4(0)	1	420
(iii) *Ant2XML*	372 (359,13)	24	1	0	0	0	0	0	0	2(0)	3	276
PetriNet to/from PathExpression	1604 (908,696)	28	42	2	1	0	4	2	0	106(5)	8	57
(i) *PetriNet2PathExp*	70 (70,0)	3	0	0	0	0	0	0	0	0(0)	1	3
(ii) *XML2PetriNet*	228 (136,92)	5	8	0	0	0	0	0	0	25(0)	2	10
(iii) *PetriNet2XML*	222 (189,33)	5	3	1	0	0	0	0	0	12(0)	4	10
(iv) *PathExp2PetriNet*	104 (87,17)	3	1	0	0	0	0	0	0	3(0)	0	3
(v) *TextualPathExp2PathExp*	643 (317,326)	7	20	1	1	0	3	2	0	49(4)	1	21
(vi) *PathExp2TextualPathExp*	337 (109,228)	5	10	0	0	0	1	0	0	17(1)	0	10

then separately links the target objects in a second phase. Cyclic dependencies between operations/non-top rules can also arise due to recursively-defined operations based upon self-associations in the source metamodel, as with *uml2Ca* and *uml2Cb*. Such situations can be reorganised to avoid recursion, by instead pre-computing the transitive closure of the self-association using the OCL *closure* operator, as in the cases of Macedo and Cunha (2016).

EPL can be an indicator of the need to apply a pattern such as Entity Splitting (Lano and Kolahdouz-Rahimi, 2014) to refactor a complex rule which creates multiple target instances, into several simpler rules creating one target each. EPL can also be reduced in some cases by introducing a new auxiliary metamodel entity to aggregate a group of separate model elements into a single element (akin to the Value Object design pattern from programming.[3])

For ETL, an implicit invocation *e.equivalent*() can be replaced by an explicit invocation of a named rule: *e.equivalent*('*R*'), or of a named operation: *e.opR*(). This may help to reduce *CBR* measures and make the specification more understandable.

Examples of MT refactorings for the ATL cases of Wimmer et al. (2012) have been given in Section 7. These involve factoring out

[3] https://www.martinfowler.com/bliki/ValueObject.html.

Table 45
Technical debt indicator evaluations for ETL cases.

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EPL*	*EFO*	*EFI*	*CC*	*CBR*	*DC*	*UEX*
uml2xsd	17 (17,0)	2	0	0	0	0	0	0	0	3(0)	0	1
In2out	19 (19,0)	1	0	0	0	0	0	0	0	1(1)	0	0
CopyFlowchart	57 (57,0)	5	0	0	0	0	0	0	0	13(4)	1	10
RSS2ATOM	88 (74,14)	9	2	0	0	0	2	0	0	18(2)	0	0
Argouml2ecore	96 (76,20)	7	2	0	0	0	1	0	0	13(2)	1	21
CopyOO	110 (110, 0)	10	0	0	0	0	2	4	0	40(8)	3	45
OO2DB	142 (121,21)	6	3	0	0	0	0	0	0	8(1)	0	6
CDO	143 (40, 103)	4	11	0	0	0	0	0	0	14(2)	1	1
MDDTIF	145 (139,6)	18	1	0	0	0	1	0	0	29(6)	1	39
(i) *Competition2TVPP*	30 (30,0)	3	0	0	0	0	0	0	0	2(0)	0	3
(ii) *CopyTVApp*	48 (48,0)	7	0	0	0	0	0	0	0	16(2)	1	21
(iii) *TVApp2Xml*	67 (61,6)	8	1	0	0	0	1	0	0	11(4)	0	15
uml2Simulink	148 (114,34)	7	6	0	0	0	2	0	0	31(3)	1	10
Flowchart2HTML	163 (163, 0)	19	0	0	0	0	0	0	0	10(1)	0	15
(i) *base*	24 (24,0)	4	0	0	0	0	0	0	0	0(0)	0	6
(ii) *equivalent*	21 (21,0)	2	0	0	0	0	0	0	0	1(0)	0	1
(iii) *greedy*	7 (7,0)	1	0	0	0	0	0	0	0	0(0)	0	0
(iv) *inheritance*	14 (14,0)	2	0	0	0	0	0	0	0	1(0)	0	1
(v) *lazy*	31 (31,0)	4	0	0	0	0	0	0	0	4(1)	0	0
(vi) *multipletargets*	32 (32,0)	2	0	0	0	0	0.	0	0	0(0)	0	1
(vii) *primary*	34 (34,0)	4	0	0	0	0	0	0	0	4(0)	0	6
QuickerMobile2Ionic	181 (59,122)	3	5	0	0	0	0	0	0	8(1)	0	0
json2sql	190 (112,78)	8	5	0	0	0	0	0	0	16(2)	1	1
TTC17Live	206 (163,43)	7	8	1	0	1	0	0	0	19(2)	0	2
(i) *Ecore2SimpleCodeDOM*	86 (50,36)	3	7	0	0	0	0	0	0	10(2)	0	0
(ii) *Ecore2SimpleCodeDOMA*	69 (69,0)	2	0	1	0	1	0	0	0	3(0)	0	1
(iii) *Ecore2SimpleCodeDOMB*	51 (44,7)	2	1	0	0	0	0	0	0	6(0)	0	1
MarketToView	235 (57,178)	5	5	0	0	0	0	0	0	10(0)	9	10
BPMN2TBP	308 (308,0)	14	0	2	0	0	1	0	0	21(3)	0	15
StateElimination (TTC 2017)	313 (155,158)	8	2	1	2	2	0	0	2	6(0)	0	4
(i) *MainTask*	229 (126,103)	4	1	1	1	2	0	0	2	3(0)	0	1
(ii) *Extension1*	84 (29,55)	4	1	0	1	0	0	0	0	3(0)	0	3
ECore2GenModel	316 (171,145)	9	7	1	0	0	0	1	0	33(1)	1	28
SQL2Java	341 (46,295)	5	12	0	0	0	0	0	1	15(0)	3	3
cdrat	344 (344, 0)	16	0	2	0	1	0	0	1	16(0)	2	26
(i) *cdrat1*	103 (103, 0)	7	0	0	0	0	0	0	0	7(0)	1	15
(ii) *cdrat2*	159 (159, 0)	6	0	1	0	1	0	0	1	7(0)	1	10
(iii) *cdrat3*	82 (82, 0)	3	0	1	0	0	0	0	0	2(0)	0	1
JEE	380 (63,217)	4	10	0	1	1	0	0	1	12(0)	1	3
AntlrAst2Intermediate	522 (114,408)	4	37	1	0	0	2	0	0	45(0)	0	1
Newsletter2Android	593 (432,161)	13	11	1	0	0	0	0	0	20(0)	3	0
Java2Html	654 (11,643)	1	11	0	3	1	0	0	1	11(0)	3	0
sdm2Representation	1011 (91,920)	3	22	0	7	7	0	0	5	25(0)	6	0

duplicated expressions into operations, and refactoring rules by introducing rule inheritance.

B.2. Time cost of technical debt

The analysis of technical debt can be extended to consider the time cost of removing flaws of different kinds. This is the approach taken by the SQALE method for source-code TD management (Letouzey and Ilkiewicz, 2012; Letouzey, 2016). SQALE involves:

- Defining a quality model (IEC/ISO, 2011), which specifies quality requirements for artefacts in a particular software language.
- Technical debt is considered to exist iff some requirement of the quality model is breached.
- A remediation action can be associated to each requirement to define how flaws related to that requirement can be corrected.
- A remediation cost is associated to each requirement, in terms of the time cost of correcting flaws related to the requirement.
- The technical debt of a software artefact is computed as the sum of the remediation costs of all flaws in the artefact.

For UML-RSDS we can adopt the quality model of Table 2, and define associated remediation actions and costs (Table 50). These costs are based on the detailed records which we kept of the *uml2Ca*, *uml2Cb* and *uml2py* developments (Lano and Alfraihi, 2018). For UML-RSDS an *internal* chaining of transformations means sequencing subtransformations τ_1, τ_2 within a transformation system τ, operating on the same source and target metamodels. In this case a new intermediate metamodel is not needed.

Table 46
Technical debt indicator evaluations for QVT-R cases.

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EPL*	*EFO*	*EFI*	*CC*	*CBR*	*DC*	*UEX*
hsm2nhsm (recursion)	48 (48,0)	5	0	0	0	0	0	0	0	5(2)	0	1
ClassModelToClassModel	85 (85,0)	3	0	0	0	0	0	0	0	4(1)	0	0
HierarchicalStateMachine2 FlatStateMachine	85 (79, 6)	3	1	0	0	1	0	0	0	3(0)	0	0
UmlToRel	98 (65,33)	2	2	0	0	0	0	0	0	3(0)	0	0
SeqToStm	104 (104,0)	4	0	0	0	1	0	0	0	4(0)	0	0
pn2pnw	115 (115,10)	5	0	0	0	0	0	0	0	8(0)	1	0
set2oset	121 (111,10)	3	1	0	0	0	0	0	0	5(1)	0	0
SeqToStmc	149 (149,0)	5	0	0	0	0	0	0	0	6(3)	0	0
bag12bag2/bag22bag1	157 (153,4)	5	1	0	0	0	0	0	0	4(1)	0	0
ER2WebML/WebML2ER	190 (158,32)	22	8	0	0	0	0	0	0	21(0)	0	0
Ecore2copyQVT	193 (193,0)	7	0	1	0	3	1	0	0	20(0)	1	2
cdrat	202 (202,0)	17	0	0	0	0	0	0	0	19(2)	2	0
UmlToRdbms	238 (226,12)	7	1	1	0	1	0	0	0	10(3)	0	0
DNF_bbox	263 (263,0)	5	0	4	0	4	0	0	0	3(0)	3	6
gantt2cpm	378 (330,48)	7	6	2	0	4	0	0	0	17(1)	2	6
DNF	396 (396,0)	9	0	4	0	4	0	0	0	10(4)	3	6
families2persons	435 (399,36)	7	6	6	0	4	2	0	2	31(0)	2	6
dag2ast/ast2dag	439 (342,97)	14	4	0	0	0	3	0	0	33(3)	0	6
f2p/p2f	462 (332,130)	13	13	0	0	0	0	0	0	37(5)	4	10
Bpmn2UseCase	532 (532,0)	23	0	1	0	6	0	2	0	42(0)	4	33
ecore2sql2	956 (799,157)	12	26	6	0	6	6	6	0	88(3)	3	0
ecore2sql3	960 (769,191)	13	25	7	0	6	5	3	0	83(3)	3	0
communication2class	1029 (392,637)	6	36	3	2	3	0	2	0	67(2)	0	6
ecore2sql1	1120 (784,336)	12	51	8	0	8	8	6	0	118(3)	5	0
RelToCore	2038 (1937, 101)	50	5	11	0	26	6	2	0	142(7)	8	10

Table 47
Technical debt indicator evaluations for UML-RSDS cases.

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EPL*	*EFO*	*CC*	*CBR*	*DC*
Tree2Graph	28 (28,0)	2	0	0	0	0	0	0	0(0)	0
family2person	45 (45, 0)	3	0	0	0	0	0	0	0(0)	2
person2family	113 (91, 22)	5	2	0	0	0	0	0	4(0)	1
calc	83 (83, 0)	4	0	0	0	0	0	0	0(0)	0
Monte-Carlo simulation	90 (61, 29)	6	3	0	0	0	0	0	3(0)	0
GMF migration	116 (34, 82)	2	5	0	0	0	0	0	6(0)	0
CDO	182 (31, 151)	4	9	0	0	0	0	0	11(2)	0
(i) *calculateRisk*	170 (19, 151)	3	9	0	0	0	0	0	11(2)	0
(ii) *deriveSectorLoss*	12 (12, 0)	1	0	0	0	0	0	0	0(0)	0
uml2rdb	184 (155,29)	5	1	0	0	0	0	0	5(0)	1
cdrat	222 (222,0)	3	0	0	0	0	0	0	0(0)	1
PetriNet to Statemachine	425 (425, 0)	10	0	1	0	0	0	0	0(0)	0
(i) *initialise*	58 (58, 0)	4	0	0	0	0	0	0	0(0)	0
(ii) *pn2sc*	310 (310, 0)	3	0	1	0	0	0	0	0(0)	0
(iii) *cleanup*	57 (57, 0)	3	0	0	0	0	0	0	0(0)	0
movies	432 (174, 258)	11	8	0	0	0	0	2	10(0)	1
(i) *task2*	28 (28, 0)	1	0	0	0	0	0	0	0(0)	0
(ii) *task3*	8 (8, 0)	0	0	0	0	0	0	0	0(0)	0
(iii) *exttask1*	23 (22, 1)	2	1	0	0	0	0	0	2(0)	0
(iv) *exttask2*	39 (39, 0)	2	0	0	0	0	0	0	0(0)	1
(v) *exttask3*	8 (8, 0)	1	0	0	0	0	0	0	0(0)	0
(vi) *exttask4*	27 (26, 1)	2	1	0	0	0	0	0	2(0)	0
(vii) *task1*	263 (7, 256)	1	6	0	0	0	0	2	6(0)	0
(viii) *couple2clique*	11 (11, 0)	1	0	0	0	0	0	0	0(0)	0
(ix) *nextcliques*	25 (25, 0)	1	0	0	0	0	0	0	0(0)	0
StatLib	525 (49,476)	12	15	0	0	0	0	0	21(0)	0
bootstrap	574 (223, 351)	14	13	0	0	0	0	0	20(3)	0
ATL2UMLRSDS	593 (180, 413)	7	9	0	1	1	0	0	14(2)	2
Simplex	664 (6, 658)	1	17	0	1	1	0	0	30(0)	0
QVT2UMLRSDS	803 (71,732)	3	16	0	1	0	0	1	24(5)	1

(*continued on next page*)

Table 47 (*continued*).

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EPL*	*EFO*	*CC*	*CBR*	*DC*
ActivityMigration	935 (21, 914)	1	13	0	2	0	1	0	16(3)	1
TTC14Live	999 (153, 846)	1	7	1	3	0	0	2	10(0)	0
Nelson–Seigal	1219 (817, 402)	26	24	0	0	0	0	0	51(0)	13
(i) *evolve*	202 (68, 134)	5	7	0	0	0	0	0	8(0)	0
(ii) *nextgeneration*	134 (84, 50)	9	3	0	0	0	0	0	5(0)	1
(iii) *initialise*	768 (591, 177)	10	11	0	0	0	0	0	34(0)	12
(iv) *test*	115 (74, 41)	2	3	0	0	0	0	0	4(0)	0

Table 48
Technical debt indicator evaluations for different versions of UML-RSDS cases.

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EFO*	*CC*	*CBR*	*DC*
uml2Ca Version 0.3	842 (735, 107)	27	12	0	1	0	0	14(4)	1
(i) *types2C*	163 (163, 0)	9	0	0	0	0	0	0(0)	0
(ii) *program2C*	182 (182, 0)	4	0	0	0	0	0	0(0)	0
(iii) *printcode*	497 (390,107)	14	12	0	1	0	0	14(4)	1
uml2Ca Version 0.9	1272 (884, 388)	51	31	0	0	0	0	37(8)	11
(i) *types2C*	177 (177,0)	11	0	0	0	0	0	0(0)	4
(ii) *program2C*	313 (274, 39)	6	2	0	0	0	0	3(1)	3
(iii) *printcode*	782 (433, 349)	34	29	0	0	0	0	34(7)	4
uml2Cb Version 0.9	5621 (904, 4717)	43	124	0	13	5	32	206(24)	39
(i) *exp2C*	4036 (107, 3929)	8	94	0	13	5	30	171(18)	38
(ii) *printcode*	1585 (797, 788)	35	30	0	0	0	2	35(6)	1
cra (2016)	1360 (438, 922)	34	56	0	0	0	6	85(0)	0
(i) *createClasses*	89 (89, 0)	5	0	0	0	0	0	0(0)	0
(ii) *refactor*	49 (49, 0)	2	0	0	0	0	0	0(0)	0
(iii) *cleanup*	11 (11, 0)	2	0	0	0	0	0	0(0)	0
(iv) *measures*	206 (32, 174)	4	6	0	0	0	2	11(0)	0
(v) *evolve*	374 (50, 324)	4	21	0	0	0	2	29(0)	0
(vi) *nextgeneration*	293 (82, 211)	10	14	0	0	0	2	23(0)	0
(vii) *initialise*	286 (84, 202)	5	14	0	0	0	0	21(0)	0
(viii) *postprocess*	18 (18, 0)	1	0	0	0	0	0	0(0)	0
(ix) *createClasses*1	34 (23, 11)	1	1	0	0	0	0	1(0)	0
cra (2017)	1635 (741,894)	36	32	0	0	0	4	50(0)	5
uml2py Version 0.4	3157(47,3110)	5	80	0	8	2	6	109(25)	3
(i) *exp2py*	2667 (0,2667)	0	55	0	8	2	6	55(14)	2
(ii) *stat2py*	230 (0,230)	0	14	0	0	0	0	35(11)	1
(iii) *model2py*	260 (47,213)	5	11	0	0	0	0	19(0)	0
uml2py Version 0.5	4846(107,4739)	8	100	12	3	4	14	112(25)	6
(i) *exp2py*	3668 (0,3668)	0	56	12	1	3	14	53(14)	3
(ii) *stat2py*	240 (0,240)	0	12	0	2	1	0	30(7)	1
(iii) *model2py*	938 (107,831)	8	32	0	0	0	0	29(4)	2

For example, *types2C* and *program2C* within *uml2Ca*. *External* chaining means defining separate transformation systems which pass model data between them via text files. In this case a defined intermediate metamodel is required. For example, *uml2Ca* and *uml2Cb*.

Factoring a transformation into n subtransformations composed in an external chain can be a substantial undertaking, potentially involving the definition of $n - 1$ intermediate metamodels to represent the data to be transferred from one transformation to its successor, in addition to a significant testing effort. Dividing a large transformation into two parts can take at least 90 min, into 3 at least 180, etc. Extracting a helper/rule in order to reduce the size or CC of an element, or eliminate duplicated code can also be time-consuming, although perhaps less so than for programming languages because of the more concise and semantically transparent nature of MT languages. We do not take account of the problem that fixing one flaw may lead to the violation of other thresholds.

As an example of this calculation, for the *uml2py* version 0.4 transformation we have a total predicted remediation cost of 1875 min (31.25 h), which is 51% of the total development effort expended on the transformation (61 h). This is considerably higher than levels found in source code (eg., between 1% and 3% in Digkas et al. (2016)) and would place this system in the highest SQALE category. Nonetheless, we consider this estimate is realistic, and indeed insufficient time was available to carry out this remediation.

The SQALE method also provides a means to prioritise TD remediation actions by associating different flaws to different software quality characteristics (for example, duplicate code impacts on testability and maintainability characteristics in particular). SQALE could be adapted to model transformation languages by adopting the general MT quality model or by adapting this to a specific language. Associated with the model, remediation actions and costs to correct quality flaws would need to be defined, as in Table 50. Correction of technical debt affecting more essential characteristics such as testability would be prioritised over correction for less critical characteristics such as maintainability.

The factors of EPL, UEX, DC, CC, CBR and EFO impact on testability, so could be prioritised for remediation. In the case of *uml2py* only the more serious cases of such flaws were corrected in the step from Version 0.4 to Version 0.5, which removed 11 of 50 flaws.

Table 49
Quality flaws for *uml2Ca* versions.

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EFO*	*CC*	*CBR*	*DC*
uml2Ca Version 0.1/0.2	116 (0, 116)	0	12	0	0	0	0	10(4)	0
uml2Ca Version 0.3	842 (735, 107)	27	12	0	1	0	0	14(4)	1
(i) *types2C*	163 (163, 0)	9	0	0	0	0	0	0(0)	0
(ii) *program2C*	182 (182, 0)	4	0	0	0	0	0	0(0)	0
(iii) *printcode*	497 (390,107)	14	12	0	1	0	0	14(4)	1
uml2Ca Version 0.4	896 (792, 107)	28	12	0	1	0	0	15(4)	2
(i) *types2C*	163 (163, 0)	9	0	0	0	0	0	0(0)	0
(ii) *program2C*	182 (182, 0)	4	0	0	0	0	0	0(0)	0
(iii) *printcode*	550 (446,104)	15	12	0	1	0	0	15(4)	2
uml2Ca Version 0.5	1593 (1034, 559)	39	27	0	1	0	0	37(11)	2
(i) *types2C*	165 (165, 0)	9	0	0	0	0	0	0(0)	0
(ii) *program2C*	181 (181, 0)	4	0	0	0	0	0	0(0)	0
(iii) *printcode*	1247 (688,559)	26	27	0	1	0	0	37(11)	2
uml2Ca Version 0.6	2108 (1059, 1049)	49	40	0	1	0	0	55(13)	4
(i) *types2C*	165 (165, 0)	9	0	0	0	0	0	0(0)	0
(ii) *program2C*	181 (181, 0)	4	0	0	0	0	0	0(0)	0
(iii) *printcode*	1762 (713,1049)	36	40	0	1	0	0	55(13)	4
uml2Ca Version 0.7	1560 (908, 652)	46	41	0	1	0	0	52(13)	1
(i) *types2C*	165 (165, 0)	9	0	0	0	0	0	0(0)	0
(ii) *program2C*	269 (269, 0)	5	0	0	0	0	0	0(0)	1
(iii) *printcode*	1126 (474, 652)	32	41	0	1	0	0	52(13)	0
uml2Ca Version 0.8	1888 (1146, 742)	48	44	2	1	0	0	56(14)	1
(i) *types2C*	165 (165, 0)	9	0	0	0	0	0	0(0)	0
(ii) *program2C*	494 (435, 59)	6	2	2	0	0	0	3(1)	1
(iii) *printcode*	1229 (546, 683)	33	42	0	1	0	0	53(13)	0
uml2Ca Version 0.9	1272 (884, 388)	51	31	0	0	0	0	37(8)	11
(i) *types2C*	177 (177,0)	11	0	0	0	0	0	0(0)	4
(ii) *program2C*	313 (274, 39)	6	2	0	0	0	0	3(1)	3
(iii) *printcode*	782 (433, 349)	34	29	0	0	0	0	34(7)	4

Table 50
Cost estimates for remediation actions (UML-RSDS).

Transformation flaw	*Remediation action*	*Approximate cost (minutes)*
ETS, ENR, ENO	Split the transformation into a chain	Internal chain: 30 each additional subtransformation External chain: 90+ each split
ERS, EHS, CC	Refactor excessively large element to extract helpers/rules	30 each case
EFO	Split a rule which has fan-out > 5	30 each case
EFI	Factor out common code from different elements which call the EFI operation	30 each case if possible
DC	Extract the cloned expression/ statement as a new helper	15 to 30 each clone copy
CBR_1, CBR_2	Re-organisation of specification structure: for CBR_2 introduce Map Objects before Links pattern or *closure* association. Replace recursion by iteration.	30 to 45 for each CBR_2 case
EPL	Split rule or introduce auxiliary entity to group parameters	30 each case

Appendix C. Tool support

We have implemented TD analysis for ATL, ETL, QVT-R and UML-RSDS using the abstract syntax (language metamodels) of each language. The analysers have been incorporated into the Agile UML tools (Fig. 2). In addition, a stand-alone analyser for ETL has been written (Fig. 16). Agile UML is available from (https://projects.eclipse.org/projects/modeling.agileuml). The stand-alone tool is available from github.com/MSharbaf/ EpsilonTransformationTechnical Debt. All of the analysed transformations can also be found at Kolahdouz-Rahimi et al. (2020b).

In the Agile UML tool, syntactic complexity c is calculated for each operation and rule, and for each complete transformation module. The measures ETS, ERS, EHS are computed based on c. The measures of fan-out, parameter length and cyclomatic complexity are also calculated for each operation/rule. The call-graph of the transformation is computed. In the case of ATL, a target element lookup *thisModule.resolveTemp*(x, v) is counted as a dependency from the rule/operation r in which it occurs to the concrete rules r' which have an input entity type equal to (or a superclass of) the entity type of x, and an output element with name v. In the case of ETL, a call *x.equivalent*() in a

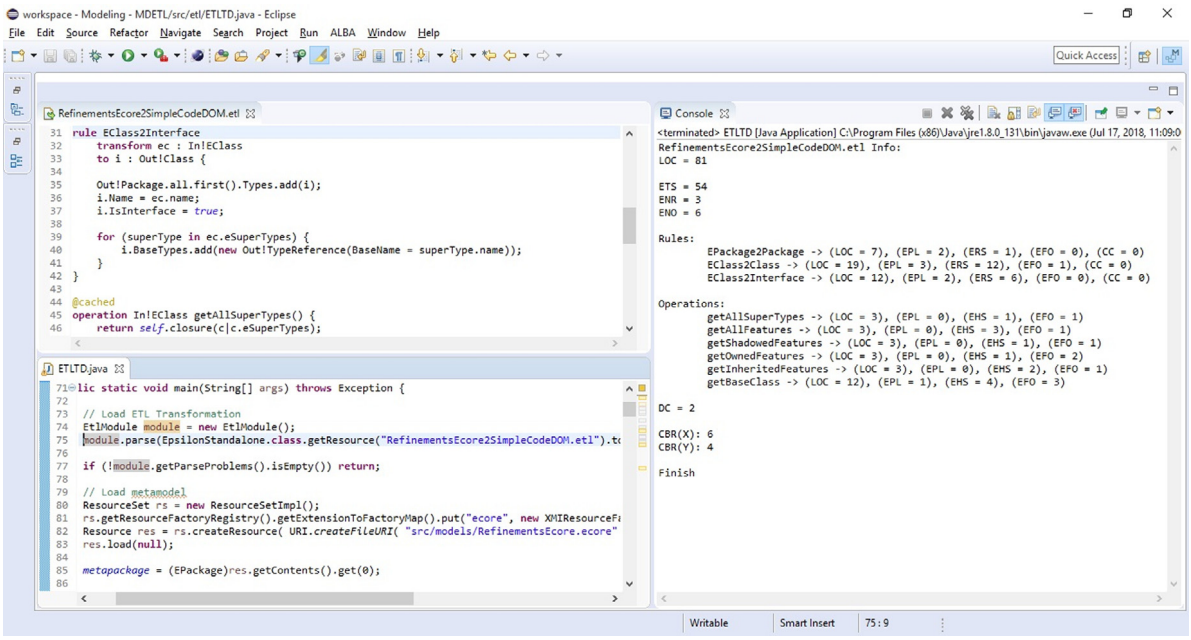

Fig. 16. Example of technical debt analysis (2).

Table 51
Technical debt indicators for industrial cases.

Transformation	*ETS* (rs, os)	*ENR*	*ENO*	*ERS*	*EHS*	*EFO*	*EFI*	*EPL*	*CC*	*CBR*	*DC*	*UEX*
KM32Measure	392 (224, 168)	6	23	2	0	3	0	0	0	52(5)	0	6
ocl2qvtp	469 (264, 205)	6	11	2	1	1	0	2	0	27(0)	10	3
ocl2qvtp_v2	460 (208, 252)	6	18	2	1	2	0	1	0	39(0)	3	0
QVTsToGraphML	196 (126,70)	6	3	0	0	0	0	0	1	17(2)	1	6
QVTs-pToQVTi	557 (442,115)	29	4	1	1	2	1	2	0	73(2)	3	351
QVTpToSchedule	651 (236,415)	9	46	0	1	2	1	1	1	84(2)	0	15
QVTmToQVTiPartition	575 (487,88)	18	4	1	0	4	0	1	0	67(0)	5	120
RelToCore	2038 (1937, 101)	50	5	11	0	6	2	26	0	142(7)	8	10
CDO	182c (31c, 151c)	4	9	0	0	0	0	0	0	11(2)	0	0
Simplex	664 (6, 658)	1	17	0	1	1	0	0	0	30(0)	0	0
Nelson–Seigal	1219c (817c, 402c)	26	24	0	0	0	0	0	0	51(0)	13	0

Table 52
Summary for industrial cases.

Transformation	*Category*	*Style*	*LOC*	$c(\tau)$	*% in rules*	*# flaws*	*flaws/LOC*	*flaws/c*
*KM32Measure**	Analysis	F	392	1015	57%	12	0.031	0.012
*ocl2qvtp**	Refinement	I	469	1477	56%	18	0.038	0.012
*ocl2qvtp_v2**	Refinement	I	460	1424	45%	11	0.024	0.008
*QVTsToGraphML**	Refinement	H	196	603	64%	5	0.0255	0.008
*QVTs-pToQVTi**	Refinement	I	557	1041	79%	16	0.028	0.015
*QVTpToSchedule**	Analysis	H	651	1412	36%	12	0.018	0.008
*QVTmToQVTiPartition**	Refinement	H	575	1268	85%	15	0.026	0.012
*RelToCore**	Refinement	H	2038	5415	95%	63	0.031	0.012
*CDO**	Analysis	F	94	182	17%	2	0.021	0.011
*Simplex**	Analysis	F	294	664	1%	4	0.014	0.006
*Nelson–Seigal**	Refinement	H	458	1219	67%	15	0.033	0.012

rule/operation r is considered to induce dependencies from r to every concrete rule with an input entity type compatible with the (statically-computed) entity type of x.

The ATL 3.1 metamodel (Eclipse, 2019) is supported, with the exception of iterative target patterns (instead, lazy and unique lazy rules are supported). For ETL, the metamodel defined in Kolovos et al. (2008) is used. The QVT 1.3 metamodel is supported, with the exception of collection templates, 'implementedby' and 'default_values' clauses (OMG, 2016).

For UML-RSDS, the cost estimate for correcting TD according to Table 50 is also provided by the tools. We separate out the costs for high-priority remediation actions to address testability flaws (UEX, EPL, DC, CC, CBR and EFO measures) and lower-priority remediation to address maintainability/changeability flaws (ETS, ENR, ENO, EFI, ERS, EHS measures).

References

Adams, S., 2017. Data-Driven Refactorings for Haskell Ph.D. thesis. University of Kent.

Alfraihi, H., Lano, K., 2018. The impact of integrating agile software development and MDD: a comparative case study. In: SAM 2018. In: LNCS, vol. 11150, pp. 229–245.

Anderson, T., Darling, D., 1952. Asymptotic theory of certain 'goodness-of-fit' criteria based on stochastic processes. Ann. Math. Stat. 23, 193–212.

Anon, 2020. Kermeta Version 2 Language Definition. http://www.kermeta.org/index_k1_k2.html.

Arendt, T., Taentzer, G., 2010. UML Model Smells and Model Refactorings in Early Software Development Phases. Technical Report FB 12, Philipps Universitat, Marburg.
Basili, V., 1992. Software Modeling and Measurement: The Goal/Question/Metric Paradigm. University of Maryland, CS-TR-2956, September.
Batot, E., Sahraoui, H., Syriani, E., Molins, P., Sboui, W., 2016. Systematic mapping study of model transformations for concrete problems. In: Modelsward. pp. 176–183.
Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A., 2019. Quality-driven detection and resolution of metamodel smells. IEEE Access.
Bonet, N., Garces, K., Casallas, R., Correal, M., Wei, R., 2018. Influence of programming style in transformation bad smells: Mining of ETL repositories. J. Comput. Sci. Educ. 28 (1).
Brambilla, M., Cabot, J., Wimmer, M., 2012. Model-Driven Software Engineering in Practice. Morgan and Claypool.
Burgueno, L., Cabot, J., Gerard, S., 2019. The future of model transformation languages: an open community discussion. JOT 18 (3).
Cabot, J., Teniente, E., 2007. Transformation techniques for OCL constraints. Sci. Comput. Program. 68, 179–195.
Correa, A., Werner, C., 2007. Refactoring OCL specifications. SoSyM 6, 113–138.
Cuadrado, J., Molina, J., 2009. Modularisation of model transformations through a phasing mechanism. SoSyM 8 (3), 325–345.
Cunningham, W., 1992. The wycash portfolio management system. In: OOPSLA 92. pp. 29–30 (addendum).
Digkas, G., Chatzigeorgiou, A., Ampatzoglou, A., Avgeriou, P., 2016. A study on the accumulation of technical debt on Framework-based web applications. In: SATToSE Seminar, July. In: CEUR Workshops Series.
Digkas, G., Lungu, M., Ampatzoglou, A., Avgeriou, P., 2017. The evolution of technical debt in the Apache ecosystem. In: ECSA. In: LNCS, vol. 10475, pp. 51–66.
Eclipse, 2019. ATL User Guide. eclipse.org.
Epsilon Generator Language, 2020. http://projects.eclipse.org/projects/modeling.epsilon.
Fleck, M., Troya, J., Kessentini, M., Wimmer, M., Alkhazi, B., 2017. Model transformation modularization as a many-objective optimisation problem. IEEE TSE 43 (11), 1009–1032.
Fowler, M., Beck, K., 2019. Refactoring: Improving the Design of Existing Code, second ed. Pearson.
Gerpheide, C., Schiffelers, R., Serebrenik, A., 2016. Assessing and improving quality of QVTo model transformations. Softw. Qual. J. 24, 797–834.
Goldschmidt, T., Wachsmuth, G., 2011. Refinement transformation support for QVT relational transformations. In: ENCS.
Halstead, M., 1977. Elements of Software Science. Elsevier North-Holland.
He, X., Avgeriou, P., Liang, P., Li, Z., 2016. Technical debt in MDE: A case study on GMF/EMF-based projects. In: MODELS.
Hebig, R., Seidl, C., Berger, T., Pedersen, J.K., Wasowski, A., 2018. MT languages under a magnifying glass. In: FSE '18. pp. 445–455.
Hermann, F., Gottmann, S., Nachtigall, N., Ehrig, H., Braatz, B., Morelli, G., Pierre, A., Engel, T., Ermel, C., 2014. Triple graph grammers in the large for translating satellite procedures. In: ICMT.
IEC/ISO, 2011. ISE/IEC 25010 Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE).
Jakumeit, E., Buchwald, S., Kroll, M., 2010. GrGen.NET: the expressive, convenient and fast graph rewrite system. InT. J. Softw. Tools Technol. Transf. 12, 263–271.
Kapova, L., Goldschmidt, T., Becker, S., Henss, J., 2010. Evaluating maintainability with code metrics for model-to-model transformations. In: Research Into Practice – Reality and Gaps. Springer.
Kolahdouz-Rahimi, S., Lano, K., Karimi, M., 2020a. Technical debt in procedural model transformation languages. J. Comput. Lang. 59, http://dx.doi.org/10.1016/j.cola.2020.100971.
Kolahdouz-Rahimi, S., Lano, K., Pillay, S., Troya, J., Van Gorp, P., 2014. Evaluation of MT approaches for model refactoring. Sci. Comput. Program. 85.
Kolahdouz-Rahimi, S., Lano, K., Sharbaf, M., Karimi, M., Alfraihi, H., 2020b. Data for investigating quality flaws and technical debt in model transformation specifications, data repository for this paper. http://dx.doi.org/10.5281/zenodo.3746606.
Kolovos, D., Paige, R., Polack, F., 2008. The epsilon transformation language. In: ICMT.
Kosti, M., Ampatzoglou, A., Chatzigeorgiou, A., Pallas, G., Stamelos, I., Angelis, L., 2017. Technical debt principal assessment through structural metrics. In: 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA, pp. 329–333.
Kusel, A., Schonbock, J., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W., 2015. Reuse in model-to-model transformation languages: are we there yet? SoSym 14, 537–572.
Lacerda, G., et al., 2020. Code smells and refactoring: a tertiary systematic review of challenges and observations. J. Syst. Softw. 167, http://dx.doi.org/10.1016/j.jss.2020.110610.
Lano, K., 2016. Agile Model-Based Development using UML-RSDS. CRC Press.
Lano, K., Alfraihi, H., 2018. Comparative case studies in agile MDD. In: FlexMDE. In: MODELS 2018.
Lano, K., Fang, S., 2020. Automated synthesis of ATL transformations from metamodel correspondences. In: Modelsward.
Lano, K., Kolahdouz-Rahimi, S., 2014. Model-transformation design patterns. IEEE Trans. Softw. Eng. 40.
Lano, K., Kolahdouz-Rahimi, S., Sharbaf, M., Alfraihi, H., 2018a. Technical debt in model transformation specifications. In: ICMT.
Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S., 2016. Solving the CRA case using UML-RSDS. In: TTC.
Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S., 2019. Declarative specification of bidirectional transformations using design patterns. IEEE Access 7, https://ieeexplore.ieee.org/document/8587240.
Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S., Sharbaf, M., 2018b. A survey of model transformation design patterns in practice. J. Syst. Softw..
Lano, K., Yassipour-Tehrani, S., Alfraihi, H., Kolahdouz-Rahimi, S., 2017. Translating from UML-RSDS OCL To ANSI C, OCL.
Letouzey, J., 2016. The SQALE Method: Definition Document Version 1.1, March. https://www.sqale.org/wp-content/uploads/2016/08/SQALE-Method-EN-V1-1.pdf.
Letouzey, J., Ilkiewicz, M., 2012. Managing technical debt with the SQALE method. IEEE Softw.
Lucio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., Syriani, E., Wimmer, M., 2016. Model transformation intents and their properties. SoSyM 15, 647–684.
Macedo, N., Cunha, A., 2016. Least-change bidirectional model transformation with QVT-R and ATL. SoSyM 15, 783–810.
Maldonado, E.da S., Shihab, E., Tsantalis, N., 2017. Using natural language processing to automatically detect self-admitted technical debt. IEEE TSE http://dx.doi.org/10.1109/TSE.2017.2654244.
Marinescu, R., 2012. Assessing technical debt by identifying design flaws in software systems. IBM J. Res. Dev. 56 (5).
McCabe, T., 1976. A software complexity measure. IEEE TSE 2, 308–320.
McCabe, T., Watson, A., 1994. Software complexity. Crosstalk 7 (12), 5–9.
OMG, 2016. MOF2 Query/View/Transformation Specification, V1.3.
Peters, R., Zaidman, A., 2012. Evaluating the lifespan of code smells using software repository mining. In: 16th European Conf. Software Maintenance and Reengineering.
Razali, N., Wah, Y.B., 2011. Power comparisons of Shapiro–Wilk, Kolmogorov–Smirnov, Lillefors and Anderson-Darling tests. J. Stat. Model. Anal. 2 (1), 21–33.
Rentschler, A., Noorshams, Q., Happe, L., Reussner, R., 2013. Interactive visual analytics for efficient maintenance of model transformations. In: ICMT 2013. In: LNCS, vol. 7909.
Shapiro, S., Wilk, M., 1965. An analysis of variance test for normality. Biometrika 52 (3–4), 591–611.
Smirnov, N., 1948. Table for estimating the goodness of fit of empirical distributions. Ann. Math. Stat. 19 (2), 279–281.
Strittmatter, M., Hinkel, G., Langhammer, M., Jung, R., Heinrich, R., 2016. Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel. In: Proc. 10th Workshop on Models and Evolution. ME 2016, pp. 30–39.
Struber, D., Acretoaie, V., Ploger, J., 2017. Model clone detection for rule-based MT languages. SoSyM 1–22.
Tom, E., Aurum, A., Vidgen, R., 2012. A consolidated understanding of technical debt. In: ECIS.
Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D., 2015. When and why your code starts to smell bad. In: ICSE.
van Amstel, M., Bosems, S., Kurtev, I., Pires, L., 2011a. Performance in model transformations: experiments with ATL and QVT. In: ICMT 2011. In: LNCS, vol. 6707, pp. 198–212.
van Amstel, M., van den Brand, M.G.J., 2011b. Model transformation analysis: Staying ahead of the maintenance nightmare. In: ICMT. pp. 108–122.
van Amstel, M., van den Brand, M., 2011c. Using metrics for assessing the quality of ATL model transformations. In: Proc. 3rd International Workshop on Model Transformation with ATL, MtATL, TOOLS 2011, pp. 20–34.
Weatherill, G.B., 1978. Elementary Statistical Methods. Chapman and Hall.
Wehaibi, S., Shihab, E., Guerrouj, L., 2016. Examining the impact of self-admitted technical debt on software quality. In: 23rd IEEE International Conference on Software Analysis, Evolution and Re-engineering.

Westfechtel, B., 2018a. Case-based exploration of bidirectional transformations in QVT relations. SoSyM 17, 989–1029.
Westfechtel, B., 2018b. Incremental bidirectional transformations: applying QVT relations to the families to persons benchmark. In: ENASE. pp. 39–53.
Westfechtel, B., Buchmann, T., 2019. Incremental bidirectional transformations: comparing declarative and procedural approaches using the families to persons benchmark. In: ENASE 2018. In: CCIS, vol. 1023, pp. 98–118.
Wimmer, M., Martinez, S., Jouault, F., Cabot, J., 2012. A catalogue of refactorings for model-to-model transformations. J. Object Technol. 11 (2), 1–40.
Zazworka, N., Vetro, A., Izurieta, C., Wong, S., Cai, Y., Seaman, C., Shull, F., 2014. Comparing four approaches to technical debt identification. Softw. Qual. J. 403–426.

Dr. Shekoufeh Kolahdouz-Rahimi is an Assistant Professor in the Software Engineering Department at the University of Isfahan. She is an active member of the Model Driven Software Engineering Research Group (MDSE) at this University. She has completed her Ph.D. in Computer Science at Kings College London in 2013. Her current research interests include model-driven software engineering and domain-specific languages.

Dr. Kevin Lano has worked for over 30 years in the fields of system specification and verification. He was one of the originators of Model-Driven Engineering, and has been a leading advocate of improving the precision of software modelling, and in applying software engineering principles to transformation construction. In recent years he has worked on the integration of model-based development and agile development. He is the author of the Eclipse Agile UML toolset for precise model-based development.

Mohammadreza Sharbaf is a Ph.D. student in the Department of Software Engineering at University of Isfahan and a member of the Model Driven Software Engineering Research Group at this University. He received his B.Sc. from the Isfahan University of Technology in 2013, and his M.Sc. from the University of Isfahan in 2016, both in Computer Engineering (Software). He is interested in Model Driven Software Engineering (Model Merging, Model Comparison and Model Transformation) and his current research is focused on conflict detection and resolution for Model Versioning systems.

Meysam Karimi is a Ph.D. student in the department of Software Engineering at University of Isfahan (UI). He received his M.Sc. from the University of Kashan, Isfahan, Iran in 2015, in Computer Engineering (Software). He is interested in Model Driven Software Engineering (MDSE) and most areas of research in computer science broadly connected to algorithms and programming languages. He is a member of Model-Driven Software Engineering Research Group (MDSERG) at University of Isfahan and his current research is focused on Model Generation in the context of model transformation testing using metaheuristic algorithms.

Dr Hessa Alfraihi is a Lecturer at Princess Nora Bint Abdulrahman University, Saudi Arabia, and a visiting research assistant at King's College London. Her research has focussed on the integration of Agile development and Model-driven development.