Science of Computer Programming 85 (2014) 5–40

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Evaluation of model transformation approaches for model refactoring

S. Kolahdouz-Rahimi, K. Lano *, S. Pillay, J. Troya, P. Van Gorp

King's College London, Strand, London, WC2R 2LS, United Kingdom

HIGHLIGHTS

- A measurement-based comparison of leading model transformation approaches on a challenging transformation case study.
- Provides a rigorous method for comparative evaluation of transformation approaches, based on quality characteristics and empirical measurement.
- A wide range of quality characteristics are evaluated, from correctness to complexity, modularity, usability and portability.

ARTICLE INFO

Article history:
Received 24 March 2012
Received in revised form 20 March 2013
Accepted 30 July 2013
Available online 13 August 2013

Keywords:
Model transformation
Measurement
Quality characteristics
Model restructuring

ABSTRACT

This paper provides a systematic evaluation framework for comparing model transformation approaches, based upon the ISO/IEC 9126-1 quality characteristics for software systems. We apply this framework to compare five transformation approaches (QVT-R, ATL, Kermeta, UML-RSDS and GrGen.NET) on a complex model refactoring case study: the amalgamation of apparent attribute clones in a class diagram.

The case study highlights the problems with the specification and design of the refactoring category of model transformations, and provides a challenging example by which model transformation languages and approaches can be compared. We take into account a wide range of evaluation criteria aspects such as correctness, efficiency, flexibility, interoperability, re-usability and robustness, which have not been comprehensively covered by other comparative surveys of transformation approaches.

The results show clear distinctions between the capabilities and suitabilities of different approaches to address the refactoring form of transformation problem.

1. Introduction

Model transformations are an essential part of model-driven engineering approaches to software development. Transformations are used to refine models from platform-independent forms to platform-specific, to migrate models in response to metamodel evolution, and generally to translate the semantic content of a model from one language to that of another. Such transformations usually have distinct source and target models and are *exogenous*, with distinct source and target languages [10]. Transformations can also be used to restructure and refactor models, in order to improve the quality of models or to make them conform to standards. Such transformations are often update-in-place, operating on a single model, and are *endogenous*, with the same source and target language. A case study of this kind is used in this paper.

As transformations have become more widely applied, they have also become large and complex software systems in their own right, to which model-driven engineering can be applied [16,25]. A range of model transformation approaches (considered as combinations of a transformation language and a tool for the language) have been developed.

* Corresponding author.
E-mail address: kevin.lano@kcl.ac.uk (K. Lano).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.07.013

Some approaches emphasise declarative specification, either logic-based or graph-theory based, others are imperative in nature, and others combine declarative and imperative aspects (hybrid approaches).

Factors such as the interoperability of transformation approaches, the efficiency of transformations implemented by means of particular approaches, and the maintainability of transformations specified in particular approaches, have become important factors in selecting an approach for a given transformation problem category.

Therefore, a suitable broad-based evaluation framework is needed to compare and assess the benefits and disadvantages of particular transformation approaches for specific categories of transformation problems. In this paper we introduce an evaluation framework based upon the software quality characteristics defined in the ISO/IEC 9126-1 and 25010 standards [7,18,19]. For each such characteristic, we use its subcharacteristics defined by [18] as *external* measures in the sense of the Goal-Question-Metric paradigm [6], and these measures will in turn be evaluated based upon quantitative *internal* measures or attributes of the transformation specification language and transformation implementation tool of each approach. For example, a subcharacteristic of functionality is *suitability*, which has quantitative measures including size, complexity, effectiveness and development effort.

As an example of applying this framework, we compare five established model transformation approaches from different language categories (GrGen: graph transformation [20], Kermeta: imperative [11], QVT-R: declarative [36], ATL: hybrid [21], UML-RSDS: general purpose MDE tool [27]) upon a transformation problem which is typical of model refactoring transformations.

Our comparison is based on the characteristics of *Functionality*, *Reliability*, *Usability*, *Efficiency*, *Maintainability* and *Portability* from ISO/IEC 9126-1. By systematically comparing and evaluating the selected transformation approaches on the case study, according to the ISO/IEC 9126-1 quality model, we can provide clear guidelines for the appropriateness of different types of transformation approaches for refactoring transformations, and the specific advantages and disadvantages of particular approaches for this type of transformation problem.

Section 2 places this research in the context of previous surveys of transformation approaches. Section 3 defines the transformation case study in detail. Section 4 defines the evaluation framework, and presents test cases. Sections 5, 6, 7, 8, 9 present the individual solutions to the problem. Section 10 compares the different solutions on the relative values of their characteristics. Section 11 gives conclusions.

2. Related work

There have been a number of publications comparing model transformation approaches on different case studies. This previous research can be divided into (i) work defining classifications for transformation approaches; (ii) work comparing approaches using subjective measures; (iii) work comparing approaches using objective measures.

A classification of model transformation approaches based on features is given in [10], which defines a general terminology for describing model transformation approaches. They cover a broad range of classification factors and present two examples of transformation.

Mens and Van Gorp in [31] applied a multi-dimensional taxonomy to categorise tools, techniques or formalisms for model transformation based on their common qualities. A number of functional and non-functional requirements for model transformations are specified.

Taentzer et al. [43] generates a taxonomy for graph transformation tools by focusing on AGG, AToM, VIATRA2 and VMTS, using the commonly-used example of transforming from class diagrams to relational databases, but without considering any quality attributes. Mohagheghi and Dehlen [34] provided an initial framework for defining and evaluating quality across different categories of model-driven engineering. This study also addressed the adaptation of the quality framework to model transformations.

In [32], quality requirements are formulated for graph transformation tools and these are analysed without focusing on a specific case study. The paper [41] also highlights several desirable features of model transformation approaches without focusing on any specific case.

In [39], Rose et al. describe the result of a migration case study at the Transformation Tool Contest 2010 workshop, with nine graph and model transformation tools applied to a model migration problem (the mapping of UML state machines to activity diagrams). All solution experts perform a peer review of the other solutions and the results are analysed statistically. Afterwards, the statistics are investigated critically by experts. The paper considers correctness and tool maturity as the most important evaluation criteria; however, these are evaluated subjectively. Their comparison is based on participant opinions and not on empirical evidence. Furthermore, important characteristics such as efficiency, complexity and modularity are not considered in this research.

A further paper based on subjective evaluations is [45], which shows the result of an earlier Transformation Tool Contest in 2007. 11 participants using graph-based tools attended the contest to perform a transformation from UML activity diagrams to formal CSP processes.

Our paper described in [28] evaluates Kermeta, Viatra, QVT-R, UML-RSDS and ATL on three case studies, a re-expression, a refinement, and the first rule of the case study considered in this paper. Subjective evaluations are given on the appropriateness of the approaches for each category of transformation.

In [1,2] some key factors that influence the internal quality of model transformations are presented and assessed by a specialised set of metrics on size, functionality, modularity and consistency.

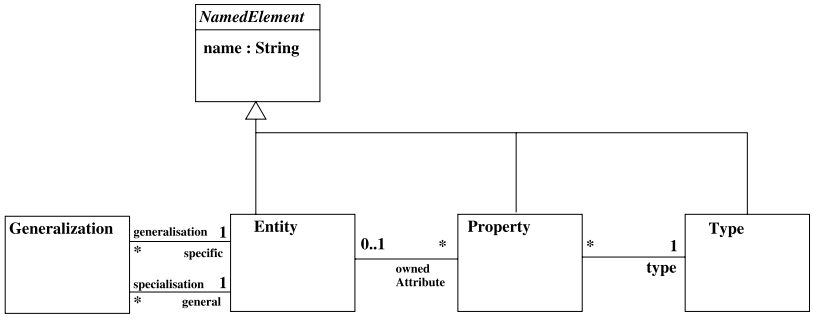

Fig. 1. Basic class diagram metamodel.

Vignaga [46] added complexity and performance to the list of quality attributes in [1]. This study did not develop metrics for different transformation paradigms and the evaluation is limited to the ATL transformation language.

Rose et al. [38] performed a comparative research study on four model migration tools (AML, COPE, Ecore2Ecore and Epsilon Flock) using an example of Petri net metamodel evolution and an evolution example involving the Eclipse Graphical Modelling Framework [14]. They conducted a single dimension research by selecting tools specialised for migration. The innovative feature of this research is the advocation of nine quality attributes for comparing migration tools, but these attributes are not based upon a standard framework.

In addition, Rose et al. [40] compared Flock to other languages for model migration, including ATL, Ecore2Ecore and COPE using a Petri net example. The comparison is focused upon the capabilities and conciseness of the approaches for specifying migrations.

In [15] three model transformation approaches (two graph-based approaches (CGT, AGG) and one hybrid approach (ATL)) are compared on a refactoring example: the removal of unstructured cycles from UML activity graphs. This example is the closest to the case study which we present here; however, the scope of our comparison is wider, including three further categories of transformation approaches, and we provide a systematic measurement-based evaluation framework to compare the approaches.

Objective measurement techniques such as dependency analysis and metamodel coverage are proposed by Van Amstel [3] as analysis techniques for model transformations. Van Amstel et al. investigate the factors influencing performance of model transformations in [4]. This paper compares the performance of particular execution engines of three transformation languages: ATL, QVT-R and QVT-O on two case studies: the UML to relational database transformation, and a tree-to-tree transformation. The effect of language constructs is also analysed by comparing different styles of implementation in ATL on the first of these case studies.

Kapova et al. in [22] evaluate the maintainability of model transformations developed in QVT-R. They generate 24 metrics for the evaluation and apply these to three different transformations.

In this paper we follow the approach of [3,4] in considering quantitative measures of model transformations, compared across different transformation languages and styles. We extend this approach by evaluating external software characteristics using relevant internal quantitative measures. We consider also a wider range of transformation approaches, and a more complex case study than those used in [4].

3. Case study description

We have selected this case study as being typical of the category of refactoring/restructuring transformations, involving the creation, deletion and relocation of elements within a single model, and requiring fixpoint iteration of prioritised rules, and fine-grained control over rule execution to achieve optimal results. For large models, computational optimisation of the transformation is also needed. Evaluation of transformation approaches on this case study should give results indicative of the suitability of the approaches for this general category of problems.

The aim of the case study transformation is to remove from a UML class diagram all cases where there are two or more sibling or root classes which all own a common-named and typed attribute, and to rationalise and amalgamate all such apparent clone copies of attributes.

It is used as one of a general collection of transformations (such as the removal of redundant inheritance, or multiple inheritance) which aim to improve the quality of a specification or design level class diagram.

Fig. 1 shows the metamodel of the source and target language of this transformation.

It can be assumed that:

Class name uniqueness: No two classes (instances of *Entity*) have the same name.
Type name uniqueness: No two types have the same name.
Property name uniqueness in classes: The owned attributes (properties) of each class have distinct names within the class, and do not have common names with the attributes of any superclass.

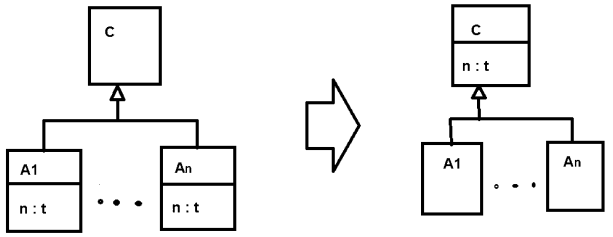

Fig. 2. Rule 1.

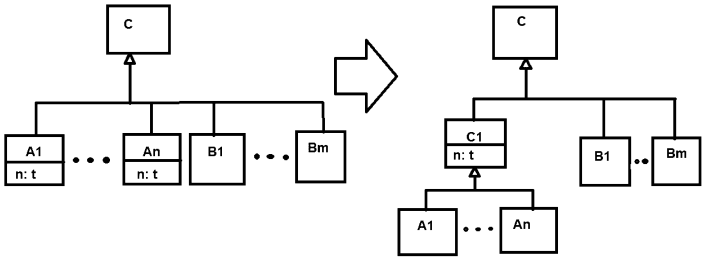

Fig. 3. Rule 2.

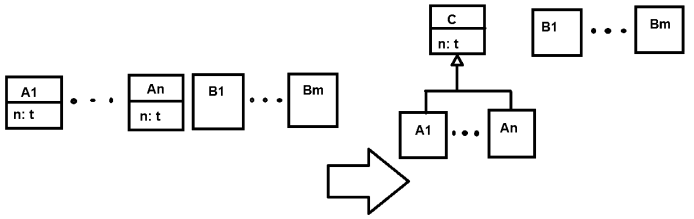

Fig. 4. Rule 3.

Single inheritance: There is no multiple or redundant inheritance.

These assumptions *Asm* must also be preserved by the transformation.

The informal transformation steps are the following:

(1) *Pull up common attributes of all direct subclasses:* If the set $g = c.specialisation.specific$ of all direct subclasses of a class c has two or more elements, and all classes in g have an owned attribute with the same name n and type t, add an attribute of this name and type to c, and remove the copies from each element of g (Fig. 2).

(2) *Create subclass for duplicated attributes:* If a class c has two or more direct subclasses $g = c.specialisation.specific$, and there is a subset $g1$ of g, of size at least 2, all the elements of $g1$ have an owned attribute with the same name n and type t, but there are elements of $g - g1$ without such an attribute, introduce a new class $c1$ as a subclass of c. $c1$ should also be set as a direct superclass of all those classes in g which own a copy of the cloned attribute. Add an attribute of name n and type t to $c1$ and remove the copies from each of its direct subclasses (Fig. 3).

(3) *Create root class for duplicated attributes:* If there are two or more root classes all of which have an owned attribute with the same name n and type t, create a new root class c. Make c the direct superclass of all root classes with such an attribute, and add an attribute of name n and type t to c and remove the copies from each of the direct subclasses (Fig. 4).

It is a requirement of the transformation to minimise the number of new classes introduced, to avoid introducing superfluous classes into the model. This means that rule 1 “Pull up attributes” should be prioritised over rules 2 “Create subclass” or 3 “Create root class”. In addition, the largest sets of duplicated attributes in sibling classes should be removed before smaller sets, for rules 2 and 3.

Table 1
ISO/IEC 9126-1 quality characteristics.

Characteristics	Subcharacteristics
Functionality	Suitability, Accuracy, Interoperability, Security, Functionality compliance
Reliability	Maturity, Fault tolerance, Recoverability, Reliability compliance
Usability	Understandability, Learnability, Operability, Attractiveness, Usability compliance
Efficiency	Time behaviour, Resource utilisation, Efficiency compliance
Maintainability	Analysability, Changeability, Stability, Testability, Maintainability compliance
Portability	Adaptability, Installability, Co-existence, Replaceability, Portability compliance

4. Evaluation framework

In this section we define an evaluation framework to evaluate and compare the selected approaches (ATL, GrGen.NET, Kermeta, QVT-R, UML-RSDS), using the ISO 9126-1 quality framework [18] and the Goal-Question-Metric paradigm [6] to systematically identify and measure different characteristics of the approaches.

4.1. ISO software quality standard 9126-1

The International Organisation for Standardization (ISO) has defined a set of ISO and ISO/IEC standards related to software quality [17–19]. For the purpose of this research we selected the ISO/IEC 9126-1 framework, which is based upon the definition of a *Quality Model* and its use for software evaluation [18]. This framework defines quality models based on general characteristics of software, which are further refined into subcharacteristics. Table 1 enumerates the six quality characteristics defined in ISO/IEC 9126-1 and their decomposition into subcharacteristics.

Subsequently, a revised standard, ISO/IEC 25010, has been issued [19], in this the Functionality subcharacteristics Security and Interoperability have been promoted to the status of top-level characteristics. Understandability has been renamed to Appropriateness, and Attractiveness as Likability.

4.2. Model transformation quality factors

Relevant characteristics and subcharacteristics for evaluation of model transformation approaches can be selected from the ISO/IEC 9126-1 and ISO/IEC 25010 quality framework. The 'software product' in this case is the transformation language and its associated tools and methods. The quality of the transformation approach not only concerns the direct quality of its language and tool support, but also the potential quality of transformations developed using the approach. We will incorporate both aspects into our evaluation framework. Our definitions of the characteristics and subcharacteristics are closely based on [18]. For the top-level characteristics of ISO/IEC 9126-1, the definitions of [18] coincide with the definitions of [19]. We decompose the characteristics and subcharacteristics into measurable attributes (such as the syntactic complexity of a specification). The first three columns of Table 2 summarise the chosen characteristics, subcharacteristics and their corresponding measurable attributes. One attribute may be related to more than one quality factor. The fourth column of the table will be explained later.

Characteristics such as interoperability and adaptability can be interpreted and evaluated in several different ways. Here, we have evaluated these characteristics based upon key factors specific to model transformations. For example, the ability to interwork with Eclipse is a key factor for the interoperability of a transformation approach.

In cases where a numeric value is not appropriate for an attribute (such as Fault tolerance, Maturity) a three-point scale is used to summarise the relative values of attributes (e.g., values $-1, 0, +1$ to denote *Low*, *Medium*, *High*). In cases where further distinctions are meaningful (abstraction level, correctness properties, and usability properties) a five-point scale is used (e.g., values $-2, -1, 0, +1, +2$ to denote *None*, *Low*, *Medium*, *High*, *Comprehensive*). Higher numerical values always represent higher quality (even for attributes such as *Complexity*, where a higher value represents lower complexity).

4.2.1. Functionality

ISO/IEC 9126-1 and 25010 define functionality as "the capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions".

For a transformation approach, we interpret functionality as the capability of the transformation approach to define and implement transformations which meet the stated and implied needs of the transformation requirements. The subcharacteristics of this quality are as follows, based on the corresponding definitions of the subcharacteristics in [18]:

Suitability: The capability of a transformation approach to provide an appropriate means to express the functionality of a transformation problem, at an appropriate level of abstraction, and to solve the transformation problem effectively and with acceptable use of resources (developer time, computational resources, etc.).

Accuracy: The capability of the transformation approach to provide a correct transformation implementation.

Interoperability: The capability of a transformation approach to support transformation composition and to be used within other transformation and software development environments.

Table 2
Selected quality characteristics for evaluating model transformation approaches.

Characteristic	Subcharacteristic	Attribute	Metric (Unit)
Functionality	Suitability	Abstraction level	five-point scale
		Size	LOC
			three-point scale
		Complexity (overall)	three-point scale
		syntactic complexity	*# operators*
			# features/entities
		structural complexity	*# calls (total)*
			# recursive calls
			max. call depth
		Effectiveness	three-point scale
		Development effort	person minutes
		Execution time	three-point scale
		Maximum capability	three-point scale
	Accuracy	Correctness	five-point scale
			syntactic (five-point)
			termination (five-point)
			confluence (five-point)
		Completeness (via Effectiveness)	three-point scale
	Interoperability	Embeddability	three-point scale
		Close to well-known notation	three-point scale
		Interoperable with Eclipse	three-point scale
Reliability	Maturity	History of use	three-point scale
			# years in use
			# solutions documented
	Fault tolerance	Tolerance of false assumptions	three-point scale
Usability	(overall)	(overall)	five-point scale
	Understandability	*Understandability (survey)*	*five-point scale*
	Learnability	*Learnability (survey)*	*five-point scale*
	Attractiveness	*Attractiveness (survey)*	*five-point scale*
Efficiency	Time behaviour	Execution time	three-point scale
		Maximum capability	three-point scale
Maintainability	Changeability	(overall)	three-point scale
		Size	three-point scale
		Complexity (overall)	three-point scale
		Modularity (overall)	three-point scale
		factorisation	*% unique expressions*
		cohesion	*% internal calls*
		coupling	*% external calls*
Portability	Adaptability	Extensibility	three-point scale

Functionality compliance: The adherence of the transformation language and tool to standards. For example, the selected approaches in this paper partially or completely use the OCL (Object Constraint Language) standard notations in their specification languages.

We evaluate suitability using a set of attributes which have quantifiable measures. The abstraction level of a transformation approach, and the relative size and complexity of specifications defined using the approach, are both factors which influence the suitability of the approach: the higher the abstraction level, the more direct it is to express transformation requirements in the transformation language, reducing the likelihood of errors being introduced at this stage. Both the size and complexity of a transformation specification also affect suitability: the larger the size and complexity, the more difficult it is to analyse the specification and to verify its correctness with respect to requirements and to verify its internal consistency. The development effort involved in specifying and implementing a transformation in an approach is a further important factor in measuring the suitability. As the amount of human resources which needs to be spent on the development of transformation tasks increases, the less suitable the approach becomes. The relative effectiveness of transformations produced by the approach in solving the transformation problem is also considered significant, together with the computational resources the transformations require to execute: the higher these requirements are, the less suitable is the approach.

Accuracy refers to the capability to achieve correctness and completeness of the transformation specification and implementation. Correctness includes ensuring the construction of target models which satisfy the language constraints of the target language (syntactic correctness), and ensuring the termination and confluence of the transformation implementation.

A complete transformation carries out all required functionality of the transformation, for all models that satisfy the transformation assumptions. In this paper we use the effectiveness of the transformation (the percentage of removable attribute clones which are actually removed by the transformation) as a completeness measure.

Interoperability means that the transformation approach should be interoperable with other model-driven development environments and tools. This is difficult to evaluate in general; however, we consider that two important factors in achieving a good level of interoperability is being able to interoperate with the widely-used Eclipse environment for MDE, and having a notation close to well-known notations such as the OCL. The transformation approach should also support the embeddability of transformations developed by the approach within larger transformation processes, developed with the same or different approaches (internal or external composability).

Security is not generally a quality of interest for model transformation approaches.

4.2.2. Reliability

This characteristic is defined as "the capability of the software product to maintain a specified level of performance when used under specified conditions" in [18,19].

Important factors for reliability of transformations are the maturity of the model transformation approach (in terms of a history of successful use), and the fault tolerance capabilities of implemented transformations produced by the approach.

Maturity is evaluated based on the number of years an approach has been publicly available and the number of case studies to which it has been successfully applied.

Fault tolerance covers how well the transformations produced by the approach handle errors within models or within processing. This is evaluated by identifying if the transformations can check the validity of models, and if they can produce error messages which identify the error location in terms of the transformation specification.

4.2.3. Usability

This is defined in [18,19] as "The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions". For transformation approaches this can be interpreted as referring to the clarity of transformation specifications defined using an approach, and the simplicity of use of the transformation tools for an approach.

Understandability: How easy is it to comprehend a transformation specification?
Learnability: To what degree can the transformation language and tool be learned in a reasonable timescale and with reasonable effort?
Attractiveness: How acceptable is the language and tool for the user?

Empirical studies with representative users and tasks are considered one of the best techniques to measure usability of software systems. In this paper we use a survey of users to measure understandability, learnability and perceived attractiveness. People with different expertise were asked to fill in an online questionnaire. The form contains five questions. The first question identifies the level of knowledge of the participant. For Understandability, the survey asks "How easy is it to relate the informal to the formal specification?" For Learnability, the survey asks "How much effort is needed to understand the transformation?" This is considered to be a learnability factor because it includes the effort required to learn sufficient elements of the transformation notation to understand the meaning of the transformation expressed in the notation. Finally, for Attractiveness, the survey asks "How well structured is the transformation specification?" as well as "How attractive is the specification notation?"

In addition to these questions related to perceived usability, we include a small test case to assess the actual understanding of the transformation by a participant: the participant needs to explain where in the code a particular aspect of the transformation (promoting a duplicated attribute to a superclass) is dealt with.

The difference between this score for actual understanding and the level of prior knowledge is also taken as a factor for actual learnability.

Operability and Usability compliance of transformation tools are not considered in this paper but the tools and case studies have been uploaded to the *SHARE* environment [24] to allow further investigation.

4.2.4. Efficiency

This characteristic is defined in [18,19] as "The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions". For transformation approaches this is the capability of the transformation approach to specify efficient transformations.

In this section we consider specifically *time behavior*, defined as "the capability of implemented transformations to provide appropriate response and processing times". The efficiency of each transformation implementation is investigated by measuring the execution time on different input model sizes. Furthermore, a series of models of increasing sizes are input to each implementation to check the maximum size of model which can be processed in a reasonable time (30 minutes or less).

4.2.5. Maintainability

In [18,19] this characteristic is defined as "the capability of the software product to be modified". We interpret this as the capability of transformation specifications and implementations in an approach to be modified. Modifications may include corrections, improvements or other changes in requirements and functional specifications. We consider specifically the subcharacteristic *Changeability*, defined as "the capability of the transformation approach to enable a specified modification of a transformation to be effectively implemented". Changeability is considered to be one of the most important properties of transformation approaches in the research community [13].

We measure changeability indirectly by measuring size, complexity and modularity. A larger sized specification generally results in a less flexible transformation, likewise for increased complexity. In addition, the greater the modularity of the specification, the easier it should be to change the transformation.

An alternative measure would be to estimate the additional development effort required to implement the change.

4.2.6. Portability

This characteristic is defined as "the capability of the software product to be transferred from one environment to another" in [18,19]. This characteristic concerns how much work is required to port a software product from one environment to another.

We consider specifically the subcharacteristic *adaptability* by considering how much effort is required to extend a transformation to a more general case of the transformation task. The evolution of source or target metamodels is a common environmental change which affects transformations, and we specifically consider how much work is required to extend the case study transformation to operate on UML 2 class diagram models.

4.3. Goal question metrics paradigm

There are many ways of identifying and defining metrics. The Goal Question Metrics (GQM) paradigm [6] is a goal-oriented approach which supports the measurement process in the software engineering domain. A measurement goal is analysed through four different dimensions.

- Object of study: includes the selection of the entity or set of entities to be investigated for the purpose of measurement.
- Purpose: the reason for investigating the entity or set of entities under study.
- Quality focus: consideration of the attribute of interest for investigation of each entity.
- Point of view: identifying the measurement which is useful.

In GQM, the goals of the measurement are identified. Several measurement goals may be pursued at the same time. Goals are refined to a set of quantifiable questions, and questions can be reused across goals. Finally, a set of metrics is associated with every question in order to answer it in a quantitative way. The same metrics may be associated to more than one question.

The GQM approach for measurement offers several advantages. For example, writing goals allows us to focus on what the important issues are. Defining questions enables us to make the goal more specific and suggests metrics that are relevant to the goals. The resulting GQM lattice allows us to see the full relationship between goals and metrics. It determines what goals and metrics are missing or inconsistent, and provides a context for interpreting the data after it is collected. In the following section we provide quantitative metrics for each of the attributes of Table 2, considered as GQM goals.

4.4. Evaluation criteria

The third and fourth columns of Table 2 show the evaluation criteria and metrics for assessing the solutions to the refactoring case. The evaluation criteria correspond to the list of measured attributes from the previous sections. These criteria are divided into properties of the transformation language (such as abstraction level, specification size, complexity, development effort, provability of syntactic correctness, extensibility, closeness to a well-known notation, understandability, modularity) and properties of an implementation in a particular tool for that language (e.g., effectiveness, execution time, termination, confluence, Eclipse interoperability, maximum capability, fault tolerance). Maturity is a feature of both aspects. The aspects should be separated because there may be several different tools for a given language (e.g., we use Medini [30] for QVT, but there are other tools for QVT).

Abstraction level is classified for the presented solutions to the problem as Very high $(+2)$ for primarily declarative solutions, High $(+1)$ for declarative solutions with a significant imperative component, Medium (0) for hybrid declarative-imperative solutions, Low (-1) for primarily imperative solutions with some declarative component and Very low (-2) for entirely imperative solutions. Declarative solutions generally express the transformation specification in ways that are close to the requirements, thus reducing the risk of an incorrect specification. Imperative specifications introduce program-level detail which increases the distance of the transformation definition from the requirements.

Size (in lines of code) is classified as Low $(+1)$, Medium (0), High (-1) according to Table 3. The ranges were chosen based on the distribution of size values for the case study solutions $(24, 81, 83, 102, 653)$: the first quartile of values is considered Low, the second and third Medium, and the 4th High. A lower LOC value corresponds to a higher quality. Even

Table 3
Size categories.

Value range	Category
0...50 lines	Low (+1)
51...375 lines	Medium (0)
376+ lines	High (−1)

Table 4
Complexity categories.

Value range	Category
0...120	Low (+1)
121...700	Medium (0)
701+	High (−1)

Table 5
Maximum capability categories.

Value range	Category
0...3000 elements	Low (−1)
3001...50 000 elements	Medium (0)
50001 + elements	High (+1)

if the ATL solution is scaled by a factor of 3 (to 243) and the QVT solution by a factor of 1.5 (to 125), to reflect that they are partial solutions, the classification of the approaches would not be changed.

Syntactic complexity is due to the complexity of the expressions used within the specification: the greater the number of expression operators (such as $=$, :, $\rightarrow$*exists*()) or references to metamodel entity types or features (such as *Generalization*, *ownedAttribute*, *specific*, etc.), the more effort is required to comprehend and work with the specification. Syntactic complexity is measured by summing the number of occurrences of operators, feature names and entity type names in the transformation definition.

Structural complexity of a transformation is measured by the number of calls, the number of recursive calls and the maximum call depth in the transformation definition.

The factors of syntactic and structural complexity are summed to provide an overall complexity measure because both are independent factors in the complexity of a specification. Table 4 gives the classification of complexity as Low (+1), Medium (0) or High (−1), based on the distribution of measured values (102, 132, 152, 190, 1214) in the case study solutions. Scaling of the ATL (to 456) and QVT-R (to 285) solutions does not change the classifications of the solutions.

The effectiveness measure used is the proportion of clone copies of attributes which are removed by the transformation. That is, if there are n copies of attributes which could, in principle, be removed by the rules of Section 3, and the implemented transformation removes $m \leqslant n$ copies, the effectiveness is m/n. Effectiveness measures of 75% or more are considered High (+1), measures of 50% to 74% are considered Medium (0), measures below 50% as Low (−1). In addition, a solution is optimal if the minimum possible number of new classes are introduced.

Development effort is measured in person-minutes. Low (+1) effort is considered to be 0 to 110 minutes, Medium (0) 111 to 360, and High (−1) over 360 minutes. These categories are based on the distribution of actual effort values in the case study solutions (Table 20). Time taken to understand the problem is not included, nor is the time taken to construct the test case models, but iterative testing using these models is included. Scaling of effort for ATL places it in the High category with Kermeta, scaling for QVT-R does not change its classification.

Execution time of the transformation implementation does not include the loading and unloading of models from the transformation tool. The execution time was measured on the common SHARE platform [44], using a virtual machine with 1 GB of main memory and one processor core assigned to it. The classification categories for execution time are based on the execution time for the model of 100 copies of test case 2: Low (+1) (under 600 ms); Medium (0) (600 ms to 500 s); High (−1) (over 500 s).

Maximum capability (to process a model within 30 minutes) is classified as Low (−1), Medium (0), High (+1) according to Table 5. This is normalised based on the actual values for the solutions (1000, 5000, 5000, 100 000, 100 000).

Correctness is divided into syntactic correctness, termination and confluence. Syntactic correctness includes not only the capability to establish the constraints of the target metamodel of a transformation (in this case the properties *Asm* of Section 3), but in addition the capability to establish or preserve correct inverse links to associations (in this case study the pairs *general*/*specialisation* and *specific*/*generalisation* of roles). The classification of correctness is given by an average of three separate 5-point measures for syntactic correctness, termination and confluence. Each measure separately is rated −2 (None), −1 (Low), 0 (Medium), 1 (High) and 2 (Comprehensive).

Interoperability consists of Embeddability: how effectively the transformation can be reused within a larger quality improvement process, consisting of transformations to (1) remove redundant inheritance, (2) remove multiple inheritance, (3) replace concrete superclasses by an abstract class and a new concrete subclass of this class. The support for embeddabil-

Table 6
Embeddability categories.

Property	Category
Both internal and external composition support	High (+1)
Only internal or external composition support	Medium (0)
No convenient mechanism for composition	Low (−1)

Table 7
Fault tolerance categories.

Property	Category
Run-time checks with exception handling	High (+1)
Capable of checking source model validity	Medium (0)
Lacks fault detection/response capabilities	Low (−1)

Table 8
Extensibility categories.

Property	Category
Extension can be done in modular manner without substantial effort	High (+1)
Possible in principle, requires substantial extension of specification	Medium (0)
Impractical due to effort required	Low (−1)

ity is classified according to Table 6. Another factor for interoperability is the closeness to a well-known notation, which is graded by a three-point scale: High (+1) for a common syntax and semantics to a well-known notation (in our examples, this is OCL); Medium (0) for a variant syntax and/or variant semantics; Low (−1) for no similarity.

Interoperability with Eclipse is given by a three-point scale: High (+1) for complete integration; Medium (0) for interoperability via exported/imported data files only; Low (−1) for no interoperation mechanism.

Maturity is considered low (−1) for languages/tools of less than 4 years public availability, medium (0) for 4 up to eight years availability, and high (+1) for more than 8 years. These ranges are normalised based on the maturity values of the case study approaches (3, 5, 7, 9, 9).

Fault tolerance (robustness) considers if there are facilities in transformations or the transformation execution environment to statically or dynamically detect and handle erroneous situations, such as detecting invalid source models where some of the transformation assumptions are false. The support for fault tolerance is classified according to Table 7.

Modularity is composed of (1) factorisation: the percentage of unique subexpressions of rules, at or above a certain minimum syntactic complexity (7 here), and (2) cohesion: the proportion of calls which are internal to modules (100% if there are no calls of any kind), versus the proportion of calls between modules (0% if no calls). These factors are not independent so are considered separately. An approach has high (+1) modularity if both cohesion and factorisation are high, low (−1) if both are low, and medium (0) otherwise. The factors of size, complexity and modularity (3-point scales) are averaged to obtain an overall maintainability score as a 3-point scale.

Extensibility considers how much effort is required to adapt the transformation to extended metamodels, specifically to the UML 2 class diagram metamodel [37]. The support for extensibility is classified according to Table 8.

For measures of size, complexity, effectiveness, efficiency, execution time, maximum capability, development effort, maturity, factorisation and cohesion we have chosen specific boundary values to give a high-level classification of measures (as Low, Medium, High, etc.). These boundary values are based on the actual distribution of values in the specific solutions, but can be varied depending on the needs of the evaluator. With simple tool support, the effect of such changes upon the overall ranking of the evaluated approaches (Table 23) could easily be visualised.

The developers of the UML-RSDS, GrGen.NET, Kermeta and ATL solutions were experts in the respective languages. The developer of the QVT-R solution had experience of 2 years in using Medini QVT. There may be some justification in weighting the evaluations of the QVT-R solution more favourably compared to the other solutions because of this factor.

4.5. Test cases

The solutions are tested on five test cases of increasing size and complexity. These test cases represent both typical scenarios which could be expected to arise in class diagram modelling (test cases 1 to 4), and pathological examples designed to check the behaviour of the transformation in extreme cases (test case 5 and the duplications of test case 2). The test cases were formulated by K. Lano and S. Kolahdouz-Rahimi in text format (for UML-RSDS) and Eclipse XML format (for the other approaches). The test models are available in an online VM [24] (SHARE: XP-TUe_SCP_ESEiC11_Quality Improvement_Update_Resubmission.vdi).

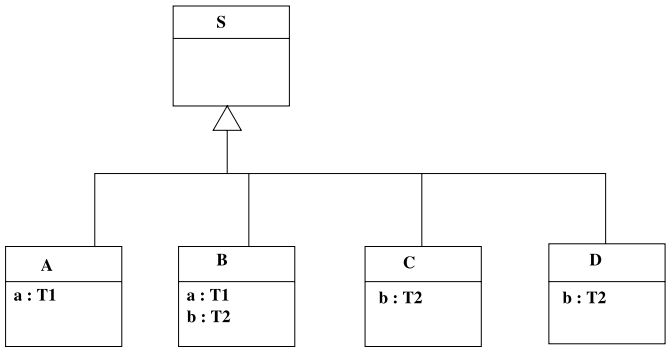

Fig. 5. Test case 1.

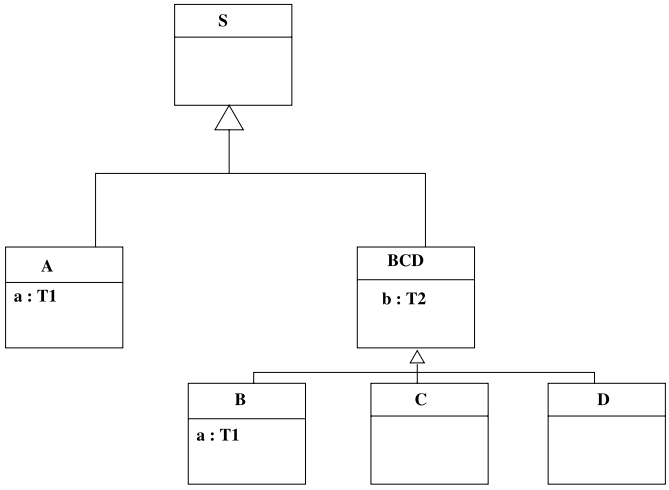

Fig. 6. Test case 1 result.

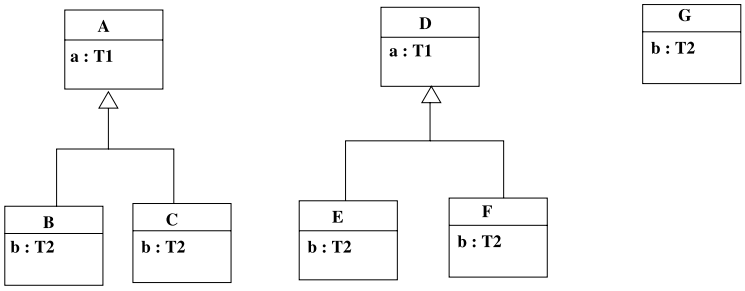

Fig. 7. Test case 2.

The first test case is a simple test for alternative applications of rule 2 “Create subclass” from Section 3. Fig. 5 shows the starting model.

Applying the rule to classes *B*, *C*, *D* is the preferred choice because it creates fewer new classes than an application to *A* and *B* followed by an application to *C* and *D*, although both solutions remove the maximum possible 2 clone attribute copies.

The resulting model should therefore have a superclass of *B*, *C* and *D* containing an attribute with name “b” and type *T*2 (Fig. 6).

A larger test case, involving applications of rules 1 “Pull up attributes” and 3 “Create root class”, is shown in Fig. 7.

The ideal result of applying the transformation to this test is shown in Fig. 8, but other results are possible in which the factoring of duplicated attributes is incomplete. For example, the sequence of rule applications:

```
b from E, F to D   (rule 1)
```

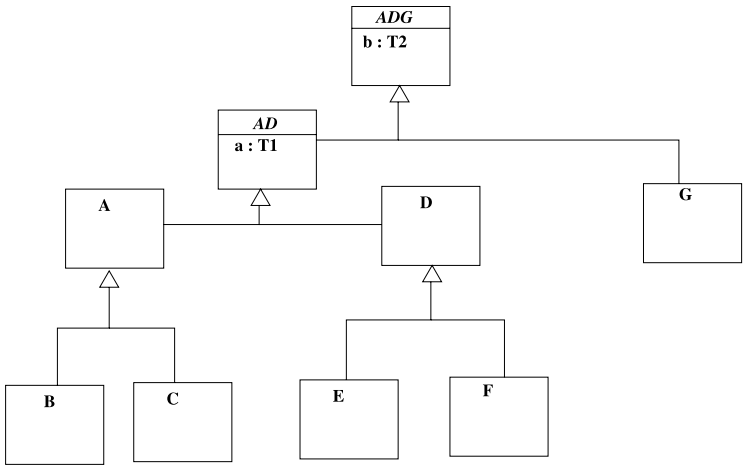

Fig. 8. Expected result of test 2.

```
b from D, G to DG  (rule 3)
b from B, C to A  (rule 1)
b from A, DG to ADG  (rule 3)
```

fails to amalgamate the clone copies of attribute a, and has 80% effectiveness.

The third test case has 20 classes and 20 properties, with 13 clone copies, and with a maximum inheritance depth of 4. A maximum of 11 clone copies can be removed, with no more than four new classes introduced.

The fourth test case has 105 classes arranged in an inheritance hierarchy of depth 3. Classes 1 to 5 are root classes, classes 6 to 30 inherit from these in groups of 5 (e.g., classes 6 to 10 inherit class 1), and classes 31 to 105 inherit the second level classes in groups of 3 (e.g., classes 31, 32 and 33 inherit from class 6). There are 12 essentially different attributes, each with 10 clones. The attribute a_k and its copies are placed successively in classes $31 + (k-1) * 5$ to $41 + (k-1) * 5$. For this test case, a maximum of 92 clones can be removed (out of 120), and a minimum of 14 new classes need to be introduced.

Test case 5 has 500 classes, each of which is a root class, and there are ten attributes in each class, with the attributes of each class being a copy of those in each other class (i.e., 5000 attributes, with 4990 clone copies). Only one new class needs to be introduced, as a superclass of all the other classes, and all redundant copies of the attributes can be removed.

In addition, we carried out 'stress testing' to measure the maximum capability of a transformation tool and implementation, in terms of the maximum size of models which a transformation tool or implementation is capable of processing. These tests were models formed from duplicated copies of test case 2 (omitting D and its subclasses), of sizes up to 10 000 copies (40 000 classes, 40 000 attributes, 20 000 generalisations).

5. UML-RSDS solution

UML-RSDS is a hybrid specification language, which defines system data by UML class diagrams, together with OCL constraints, and defines behaviour by operations of classes and by use cases which operate on the class diagram. Each use case can be declaratively specified by a set *Asm* of preconditions, and a sequence *Post* of postconditions. The use case should ensure *Post* at its termination, if *Asm* holds at its initiation.

UML-RSDS is a general-purpose specification and design language for model-driven development, with a software tool which produces executable Java implementations of UML-RSDS models [27]. Transformations can be specified in UML-RSDS in terms of the source and target metamodels they operate upon (represented as class diagrams) and the transformation functionality (represented as constraints of use cases, operations, activities or other UML behavioural elements). The most abstract and declarative style of transformation specification in UML-RSDS is in terms of use cases defined by OCL constraints. We will use this style of specification for the case study.

5.1. Case study specification in UML-RSDS

The case study transformation specification in UML-RSDS consists of the class diagram of Fig. 1, and a single use case which represents the transformation. The use case has precondition constraints expressing the assumptions *Asm* of the transformation, and a sequence *Post* of three postcondition constraints ($C1$), ($C2$), ($C3$) corresponding to the three informal rules. Each of these operates on instances of *Entity*:

$$
\begin{aligned}
&(C1):\\
&a : specialisation.specific.ownedAttribute \ \&\\
&specialisation.size > 1 \ \&\\
&specialisation.specific \rightarrow forAll(\\
&\quad ownedAttribute \rightarrow exists(b|b.name = a.name \ \& \ b.type = a.type)) \ \Rightarrow\\
&\quad\quad a : ownedAttribute \ \&\\
&\quad\quad specialisation.specific.ownedAttribute \rightarrow select(\\
&\quad\quad\quad\quad name = a.name) \rightarrow isDeleted()
\end{aligned}
$$

This specifies that an instance (*self*) of *Entity*, and instance a of *Property* match the constraint LHS if: (i) a is in the set of attributes of all direct subclasses of *self*, (ii) there is more than one direct subclass of *self*, and (iii) every direct subclass of *self* has an attribute with the same name and type as a.

The conclusion specifies that (i) the property a is moved up to the superclass *self*, (ii) all other attributes with name $a.name$ are deleted from all direct subclasses of *self*.

$s \rightarrow isDeleted()$ is a built-in operator of UML-RSDS, which deletes the object or set of objects s from their model, removing them from all entity types and association ends.

Rule $(C1)$ is applied first, to the class A with subclasses B and C, when this transformation is executed on the second test case (Fig. 7), moving the copy of attribute b from class B up to class A, and deleting the copy of b in C.

$$
\begin{aligned}
&(C2):\\
&a : specialisation.specific.ownedAttribute \ \&\\
&v = specialisation \rightarrow select(\\
&\quad specific.ownedAttribute \rightarrow exists(b|b.name = a.name \ \& \ b.type = a.type)) \ \&\\
&v.size > 1 \quad \Rightarrow\\
&\quad Entity \rightarrow exists(e|e.name = name + \text{“\_2\_”} + a.name \ \&\\
&\quad\quad a : e.ownedAttribute \ \&\\
&\quad\quad e.specialisation = v \ \&\\
&\quad\quad Generalization \rightarrow exists(g|g : specialisation \ \& \ g.specific = e)) \ \&\\
&\quad\quad v.specific.ownedAttribute \rightarrow select(name = a.name) \rightarrow isDeleted()
\end{aligned}
$$

The assumption specifies that an instance (*self*) of *Entity*, and instance a of *Property* match the constraint LHS if: (i) a is in the set of attributes of all direct subclasses of *self*, (ii) the set v of all specialisations of *self* whose class contains a clone attribute of a has size greater than 1.

The conclusion specifies that: (i) a new class e is created, and the property a is moved up to e, (ii) the specialisations of e are v, (iii) e is made a subclass of *self*, and (iv) all the clone copies of a in the classes of $v.specific$ are deleted.

This rule is applied first, to class S with v referring to subclasses A and B, and operating on attribute a, when this transformation is executed on the first test case (Fig. 5). The rule creates a new subclass e of S and moves the copy of attribute a from class A up to class e, and deletes the copy of a in B. The rule is then applied to the subclasses C and D of S to promote attribute b to another new subclass $e2$ of S.

$$
\begin{aligned}
&(C3):\\
&a : ownedAttribute \ \&\\
&generalisation.size = 0 \ \&\\
&v = Entity \rightarrow select(generalisation.size = 0 \ \&\\
&\quad ownedAttribute \rightarrow exists(b|b.name = a.name \ \& \ b.type = a.type)) \ \&\\
&v.size > 1 \quad \Rightarrow\\
&\quad Entity \rightarrow exists(e|e.name = name + \text{“\_3\_”} + a.name \ \&\\
&\quad\quad a : e.ownedAttribute \ \&\\
&\quad\quad v.ownedAttribute \rightarrow select(name = a.name) \rightarrow isDeleted() \ \&\\
&\quad\quad v \rightarrow forAll(c|Generalization \rightarrow exists(g|\\
&\quad\quad\quad\quad g : e.specialisation \ \& \ g.specific = c)))
\end{aligned}
$$

This constraint matches against a class *self* and an attribute a, if: (i) *self* is a root class, (ii) a is an attribute of *self* and has at least one clone in the attributes of other root classes. v is the set of all root classes with a copy of a.

The effect of the constraint is then: (i) to create a new (root) class e, (ii) to move a into the attributes of e, (iii) to delete all clones of a from elements of v, (iv) to make the elements of v direct subclasses of e.

On test case 2, this rule is applied to classes A and D to create a new superclass AD, and to move attribute a up to this class.

The design and implementation of these constraints is automatically synthesised by UML-RSDS, following the process described in [26]. The priority ordering for the constraint implementations is based on the textual ordering of the three constraints, so that the generated design carries out a fixpoint iteration of constraint 1, then of constraint 2, then of constraint 3, all in a composite fixpoint iteration:

Table 9
Evaluation results based on the characteristics of Table 2.

Characteristic	Attribute	Unit	RSDS	GrGen	Kermeta	ATL	QVT-R
Functionality (Suitability)	Abstraction level	5PS	+2	+1	−2	0	+2
	Size	LOC	24	102	653	81	83
		3PS	+1	0	−1	0	0
	Complexity (overall)	3PS	0	+1	−1	0	0
	syntactic complexity	*# operators*	78	40	549	69	100
		# features	54	37	605	75	77
	structural complexity	*# calls (tot)*	0	23	54	6	11
		# rec c.	0	0	2	0	0
		max. cd.	0	2	4	2	2
	Effectiveness	3PS	+1	+1	+1	−1	−1
	Development effort	pers. min.	120	100	440	150	280
	Execution time	3PS	+1	0	0	−1	0
	Maximum capability	3PS	+1	+1	0	−1	0
(Accuracy)	Correctness (overall)	5PS	0	0	+1	0	−1
	syntactic	*5PS*	+1	0	0	0	−1
	termination	*5PS*	+2	+2	+1	+1	0
	confluence	*5PS*	−2	−2	+1	−2	−2
	Completeness	3PS	+1	+1	+1	−1	−1
	Embeddability	3PS	+1	+1	+1	+1	0
(Interoperability)	Close to well-known not.	3PS	+1	0	+1	+1	+1
	Interoperable w/Eclipse	3PS	0	+1	+1	+1	+1
Reliability (Maturity)	History of use	3PS	−1	+1	0	+1	0
		# YIU	3	9	7	9	5
		# SD	10	50	40	100	20
(Fault tolerance)	Tolerance of false asm.	3PS	0	0	+1	+1	+1
Usability	(overall)	5PS	0	0	0	0	−1
	Understandability	*5PS*	−1	0	0	0	−1
	Learnability	*5PS*	−1	0	0	0	−2
	Attractiveness	*5PS*	0	0	−1	0	0
Efficiency (Time behaviour)	Execution time	3PS	+1	0	0	−1	0
	Maximum capability	3PS	+1	+1	0	−1	0
Maintainability (Changeability)	(overall)	3PS	0	0	−1	0	0
	Size	3PS	+1	0	−1	0	0
	Complexity (overall)	3PS	0	+1	−1	0	0
	Modularity (overall)	3PS	0	0	−1	+1	0
	factorisation	*% UE*	68	65	53	72	61
	cohesion	*% IC*	100	100	56	100	91
	coupling	*% EC*	0	0	44	0	9
Portability (Adaptability)	Extensibility	3PS	0	0	0	−1	−1

$$\big((stat(C1)*; stat(C2))*; stat(C3)\big)*$$

where $stat(Cn)$ implements Cn. In other words, all possible applications of $(C1)$ are performed before any application of $(C2)$, and $(C3)$ is only applied when there are no possible applications of $(C1)$ or $(C2)$.

Some optimisations can be carried out (with human guidance of the synthesis process): since the conclusion of each constraint contradicts its antecedent, testing that the conclusion holds for elements that match the antecedent is superfluous and can be omitted. In $(C2)$ a *let* definition has been used (the variable v) in order to avoid repeated computation of the expression defining v. Likewise in $(C3)$.

The inverse directions of associations are automatically set when one direction is set: in $(C2)$ the update $e.specialisation = v$ automatically removes the elements of v from $self.specialisation$. Likewise in $(C3)$ the assignment $a : e.ownedAttribute$ removes a from $self.ownedAttribute$. g is added to $e.generalisation$ when $e.specific = g$ is set.

5.2. Evaluation properties

Table 9 summarises the evaluated attributes of the transformation language and tools for the UML-RSDS solution, and for the other solutions. The data in this table has not been scaled to reflect the smaller scope of the ATL and QVT-R solutions: scaling only makes a difference to one classification (of −1 instead of 0 for ATL in terms of development effort).

5.2.1. Transformation language properties

The language is capable of a very high level of abstraction in terms of logical formulae, and a primarily declarative specification can be given for this case study, although the succedents of the constraints have a procedural character, with precise orders of actions being specified.

There are no invocations of operations within the specification, so the structural complexity is 0. However, the specification fails to be modular because there are repeated occurrences of duplicated expressions. For example, the test

$$ownedAttribute \rightarrow exists(b|b.name = a.name \ \& \ b.type = a.type)$$

occurs three times, once in each constraint antecedent, and should be factored out into a query operation *hasAttribute*(*n* : *String*, *t* : *Type*) : *Boolean* of *Entity*.

The UML-RSDS language provides the capability for such factoring by the use of operations invoked from constraints. There are 60 expressions of complexity 7 or more in the rules, and 19 non-unique subexpressions of complexity 7 or more, so the percentage of such unique subexpressions is 68%. Cohesion is 100%, as there are no calls within the specification.

Development effort for each constraint was about 30 to 50 minutes, not including test case construction.

UML-RSDS has comprehensive support for the proof of syntactic correctness, using a translation to the B formal method and proof tool [29]. The transformation does not ensure that new classes have unique names, a more complex naming scheme would be needed to ensure this, or an additional assumption included that character "_" does not occur in any element name in the source model. It can be shown that the transformation does not introduce multiple inheritance: in ($C2$) the original direct subclasses *v.specific* of *self* are set instead to be direct subclasses of the new class *e*, and this is set to be a direct subclass of *self*. In ($C3$) the new class *e* is a root class with direct subclasses the set *v* of original root classes.

Attributes have distinct names within classes in the target model, because for ($C1$) there cannot already be an attribute of *self* with name *a.name*. For ($C2$) and ($C3$) the attribute is added to a new empty class.

The transformation can be included in the sequence (1), (2), (3) of quality improvement transformations defined in Section 4.4, some of its assumptions are established by transformations (1) and (2), so it must sequentially follow these transformations, it also does not interfere with their effect, so this sequential composition is valid. Likewise, transformation (3) should sequentially follow the transformation: (3) does not invalidate the transformation because the new classes introduced by (3) are all empty (they have only inherited features), so no new cases of duplicated attributes are introduced. The transformations can be composed internally in UML-RSDS by means of use case inclusion. They could also be composed externally by successive invocation of the Java programs produced for each transformation.

Although the UML-RSDS specification language is closely based upon UML notations, the language and tool only have a relatively short history of application to transformation problems (3 years).

In the usability survey, 3 of 5 respondents considered that the specification was well-structured and that low or medium effort was required to understand the specification. However only one respondent correctly identified the part of the specification responsible for moving duplicated attributes.

The solution can be used for extended class diagram metamodels; however, problems arise with the use of object creation: if the objects have more meta-attributes or features in the extended metamodel, then these will not be set by the creation action as written. For example, *Entity* may gain an attribute *isAbstract* to identify if it is abstract or not: the classes created by ($C2$) and ($C3$) must set this attribute to *true*. UML-RSDS does not provide the capability to implicitly extend constructors, except by the use of default initial values set in the metamodel. From the viewpoint of flexibility, it is preferable to retain source model objects, rather than to copy or recreate them, in order to avoid the problem of omitting the setting of their features. For example, because property object *a* is moved, rather than recreated, in the rules, the rules work for extended versions of *Property* without change, with respect to the creation/relocation of properties. The transformation can be extended to consider properties which are association member ends by making *Entity* a subclass of *Type* and permitting multi-valued attributes of class type, as in [37]. Clones of attributes would only be amalgamated if they had the same multiplicities, in addition only non-static attribute copies should be amalgamated. To make the transformation more generic, the predicate checking the equality of attributes ($b.name = a.name \ \& \ b.type = a.type$) should be replaced in all three constraints by a query *b.equals*(*a*) defined locally in *Property*.

5.2.2. Transformation implementation properties

The Java implementation is correct-by-construction with respect to the specification. Termination can be shown by using a variant function defined as the total number of *Property* instances in the model. Each constraint application reduces this measure by at least one.

However, the rules are not confluent: different choices in the ordering of applications of the rules to elements will result in different models. They are not optimally effective, since classes may be considered in any order, resulting in some possible rationalisations of attributes being omitted. It is not possible to declaratively specify fine-grained control over the order of processing elements by a rule (postcondition constraint). In ($C2$) for example, we need to iterate through the subclass attributes *a* of a class in descending order of the size of the set of subclasses that contain a copy of *a*. Only by using a more explicit style similar to that of the Kermeta solution, could such control be enforced.

Despite the non-determinacy in the execution of the transformation, it is possible to reason that certain general properties hold true for the transformation; e.g., that each class in the source model is also a class in the target model:

Table 10
Test case results for UML-RSDS.

Test case	Number of classes	Number of attributes	Execution times (ms)	Effectiveness
1	5	5	0	100% (non-optimal)
2	7	7	0	100% (optimal)
3	20	20	10	91% (non-optimal)
4	105	132	50	100% (non-optimal)
5	500	5000	96568	100% (optimal)

Table 11
Maximum capability test results for UML-RSDS.

Test case	Number of classes	Number of attributes	Execution times (ms)	Effectiveness
$2*100$	400	400	90 ms	100%
$2*200$	800	800	330 ms	100%
$2*500$	2000	2000	2363 ms	100%
$2*1000$	4000	4000	13 s	100%
$2*5000$	20000	20000	156 s	100%
$2*10000$	40000	40000	1137 s	100%

Entity@pre $\subseteq$ *Entity*. In particular, the set of leaf classes remain unchanged by the transformation, and the set of attribute names and types in each leaf class are preserved.

On the first test case the two occurrences of *a* are merged first, instead of the three occurrences of *b*, so that two new classes are created, a non-optimal result. The second test case results in the correct model, as shown in Fig. 8. Alternative input formats of the model could produce non-optimal results, however. On the third test case the transformation takes 10 ms, and removes 10 of 11 clone copies in 10 steps, introducing 4 new classes. Its effectiveness is $10/11 = 91\%$. On the fourth test case the transformation takes 50 ms and applies constraint ($C1$) 39 times, and constraint ($C2$) 16 times. It removes 92 clone copies, so has effectiveness 100%, but is non-optimal (2 additional classes are introduced). On the fifth test case the transformation takes 97 s and eliminates all the cloned attribute copies, with only 1 new class being introduced, an optimal result.

Table 10 shows the execution times (on the SHARE platform) and effectiveness measures for the test cases considered. No special optimisations were performed, such as inlining of calls in the implementation.

In general, the efficiency and effectiveness are high for class diagrams of the structure and complexity likely to be encountered in most developments (the first 4 test cases). The Java implementation generated by the UML-RSDS tools shows an above-linear growth of execution time when constructing large sets of elements, as in rule 3 in the final test case.

Table 11 shows the results of the maximum capability tests on the SHARE platform. The primary cause in the time complexity explosion are the applications of rule 3, involving the construction and search of sets of *Entity* instances of size *dim*, the number of duplications of test case 2. The results in each case were optimal.

UML-RSDS can express assumptions of a transformation, and check these when a model is loaded. It has no capability for handling runtime errors, these simply cause Java exceptions.

UML-RSDS is relatively immature, having been publicly available for two years. About 10 case studies have been published. There is no interface to Eclipse; instead, model input and output is by means of text files. There is a facility to export and import metamodel and model data as XML files, for interoperation with Eclipse.

6. GrGen.NET solution

GrGen.NET [20] is a graph rewrite system developed at Karlsruhe University, Germany. It combines declarative graph transformation rules with imperative features for the detailed programming of complex transformations. The basic units of a GrGen.NET transformation are transformation rules, which operate on one or more model elements, and which use graph patterns in the abstract syntax of the source or target metamodels to identify relevant elements to which the rule should be applied. The rewrite part of a rule specifies changes to models to be carried out for each matching group of elements. These changes can be element creation, deletion or feature modifications.

6.1. Case study specification in GrGen.NET

This section describes the GrGen.NET solution to the case study.

It is required to minimise the number of new classes introduced, i.e., to prioritise rule 1 over rules 2 or 3. This can be formalised in GrGen syntax as follows:

```
1 xgrs (rule1 || rule2 || rule3)*
```

The above script executes the three rules such that *rule2* is only executed if rule1 fails (and similarly for rule3 and rule2). The * operator iterates as long as at least one of the three rules matches.

Rule 1 “Pull up attributes” can be formally specified in GrGen syntax as follows, where ‘*Class*’ is used instead of ‘*Entity*’ for consistency with UML:

```
1 rule rule1 {
2   c:Class;
3   :SuperOf(c,g1); :SuperOf(c,g2);
4   g1:Class -:ownedAttribute-> a1:Property -:type-> t:Type;
5   g2:Class -:ownedAttribute-> a2:Property;
6   :SameAttribute(a1,a2);
7   negative {
8     g3:Class;
9     :SuperOf(c,g3);
10    g1;
11    negative {
12      g3 -:ownedAttribute-> a3:Property;
13      :SameAttribute(a1,a3);
14    }
15  }
16  modify {
17   c -:ownedAttribute-> a4:Property -:type-> t;
18   eval {
19     a4._name = a1._name;
20   }
21   exec(RemoveAttributeFromSubclasses(c, a4) ;> [createInverseEdges]);
22   }
23 }
```

Line 2 of the above listing declares that *rule1* requires the presence of a class *c*. To ensure that *c* has at least two subclasses, line 3 includes the helper pattern with name *SuperOf* twice. The first pattern call (i.e., *SuperOf(c,g1)*) binds *g1* as a subclass of *c* while the second one does the same for *g2*.

Line 4 encodes a pattern consisting of a node of type *Class*, an edge of type *ownedAttribute*, a node of type *Property* and an edge of type *Type*. The node of type *Class* is bound to *g1* (i.e., the first subclass of *c*) and the node of type *Property* is bound to *a1*. Clearly, this node represents an attribute of *g1*. Following the same rationale, node *t* represents the type of that attribute.

Line 5 is very similar to line 4 but instead of *t:Type* it contains nothing. The type constraint for attribute *a2* (i.e., the attribute of the second subclass) is handled by *SameAttribute(a1,a2)* on line 6. Besides checking that *a1* and *a2* have the same type, the helper pattern call checks that these attributes have the same name (see below for the definition of *SameAttribute*).

In summary, lines 2 to 6 encode a pattern of a class with two subclasses with the same attribute. This corresponds to the “*two or more elements*” from the informal rule description (Section 3). In the following, we clarify how it is ensured that *all* subclasses of *c* have that attribute.

The GrGen language does not have a universal qualifier for patterns in the left-hand side of a rewrite rule. However, all universally quantified constraints can be easily rewritten into negations of the corresponding existential constraints. For *rule1*, we have to rephrase the informal rule description saying “*all classes in g have...*” into “*there is no class in g that does not have...*”. Lines 7 to 15 formally express that there should not be a third subclass *g3* that does not have an attribute equal to *a1*. Notice that we conveniently reuse the helper pattern *SameAttribute* on line 13.

The *modify* block between lines 16 and 22 formalises the side-effects of *rule1*. The pattern on line 17 denotes the creation of a new attribute *a4*, which has the type *t* of *a1*. The *eval* expression between lines 18 and 20 assigns the name of *a1* to *a4*. Finally, line 21 ensures that the copies of *a1* are removed from all subclasses of *c* (see below for the definition of helper *RemoveAttributeFromSubclasses*).

Helper patterns “SameAttribute” and “SuperOf” are defined as follows:

```
1 pattern SameAttribute(a1:Property,a2:Property) {
2   independent {
3     if { a1._name == a2._name; }
4     a1 -:type-> t:Type <-:type- a2;
5   }
6 }
7 pattern SuperOf(c1:Class, c2:Class) {
8   c1 -:specialisation-> :Generalization -:specific-> c2;
9 }
```

Helper pattern *SameAttribute* contains an attribute constraint (“if...”) as well as a conventional graph pattern. Both are embedded in a so-called *independent* block. The use of *independent* is needed to allow that the edges of type *type* as well as the node of type *Type* (both on line 4) of *SameAttribute* are treated as completely free variables. Without the *independent* block, the GrGen engine would forbid that these pattern elements would be matched by host graph elements that were already matched elsewhere (which is undesirable since among others the *Type* element is typically already matched in the context of line 4 of *rule1*). In graph transformation jargon, the *independent* keyword overrides the default isomorphic matching by homomorphic matching.

Helper pattern *SuperOf* imposes that between a superclass *c1* and a subclass *c2* there is a path consisting of an edge of type *specialisation*, a node of type *Generalization* and an edge of type *specific*.

rule1 also relies upon helper rules *RemoveAttributeFromSubclasses* and *createInverseEdges*. The former is defined as:

```
1 rule RemoveAttributeFromSubclasses(c:Class, attr:Property) {
2   iterated {
3     c -:specialisation-> :Generalization -:specific-> g:Class -:ownedAttribute-> a:Property;
4         :SameAttribute(a,attr);
5         modify {
6           delete(a);
7         }
8   }
9 }
```

The *iterated* block between lines 2 and 8 of *RemoveAttributeFromSubclasses* ensures that the embedded rewrite constructs are applied as long as matches are found (instead of applying these constructs just for one match). Lines 3 to 4 of *RemoveAttributeFromSubclasses* resemble strongly the first lines of *rule1*. The two lines match all attributes from subclasses of *c* that are equal to attribute *attr*. The *modify* block between lines 5 and 7 deletes such attributes from these subclasses.

Helper rule *createInverseEdges* simply creates the inverse of those edges that do not yet have a reverse edge. Alternatively, the reverse edges could have been created explicitly in the *modify* part of the above rule definition. That would have been slightly better for runtime performance but would probably compromise the maintainability of the rule (computer cycles are cheaper than human debugging time).

The informal specification of *rule2* ("Create subclass") is different from that of *rule1* in that it deals with the case where one of the subclasses of *c* does not contain the attribute that is replicated across other subclasses of *c*. This following GrGen specification is similar to that of *rule1* in that again *g1* and *g2* are defined as subclasses with a common attribute:

```
1 rule rule2 {
2   c:Class;
3   :SuperOf(c,g1); :SuperOf(c,g2);
4   g1:Class -:ownedAttribute-> a1:Property -:type-> t:Type;
5   g2:Class -:ownedAttribute-> a2:Property;
6   :SameAttribute(a1,a2);
7
8   g3:Class;
9   :SuperOf(c,g3);
10  negative {
11  g3 -:ownedAttribute-> a3:Property;
12  :SameAttribute(a1,a3);
13  }
14  modify {
15   c -:specialisation-> :Generalization -:specific-> c1:Class;
16   c1 -:ownedAttribute-> a4:Property -:type-> t;
17   eval {
18     a4._name= a1._name;
19  }
20  exec(AddIntermediateClassAndRemoveAttributeFromSubclasses(c, a4, c1) ;> [createInverseEdges]);
21  }
22 }
```

The interesting (i.e., unique) part of the above listing is between lines 8 and 13. This part of the rules left-hand side specifies that there should be a subclass *g3* of *c* that does not have an attribute equal to *a1*.

The *modify* block of *rule2* creates a new subclass *c1* of *c* (cf., line 15) and adds an attribute equivalent to *a1* to *c1* (cf., line 16). Making *c1* the new superclass of the former subclasses of *c* is handled by helper rule *AddIntermediateClassAndRemoveAttributeFromSubclasses*. The rule also removes the replicated attribute from these former subclasses:

```
1 rule AddIntermediateClassAndRemoveAttributeFromSubclasses(super:Class, attr:Property, intermediate:Class) {
2   iterated {
3     super -:specialisation-> gOld:Generalization -:specific-> g:Class -:ownedAttribute-> a:Property;
4     intermediate; // <> super <> g
5     :SameAttribute(a,attr);
6     modify {
7       delete(gOld);
8       intermediate -:specialisation-> gNew:Generalization -:specific-> g;
9       delete(a);
10      }
11    }
12  }
```

Note that the above rule only affects those subclasses that contain the replicated attribute (because of the pattern on line 3). Other subclasses are left intact. The rule does not guarantee that the largest collection of duplicated attributes in subclasses is matched, so the result for test case 1 can be non-optimal, as for UML-RSDS.

Rule 3 "Create root class" can be formally specified in GrGen syntax as follows:

```
1 rule rule3 {
2   :IsRoot(g1); :IsRoot(g2);
3   g1:Class -:ownedAttribute-> a1:Property -:type-> t:Type;
4   g2:Class -:ownedAttribute-> a2:Property;
5   :SameAttribute(a1,a2);
6   modify {
7    c:Class -:ownedAttribute-> a3:Property -:type-> t;
8    eval {
9      a3._name= a1._name;
```

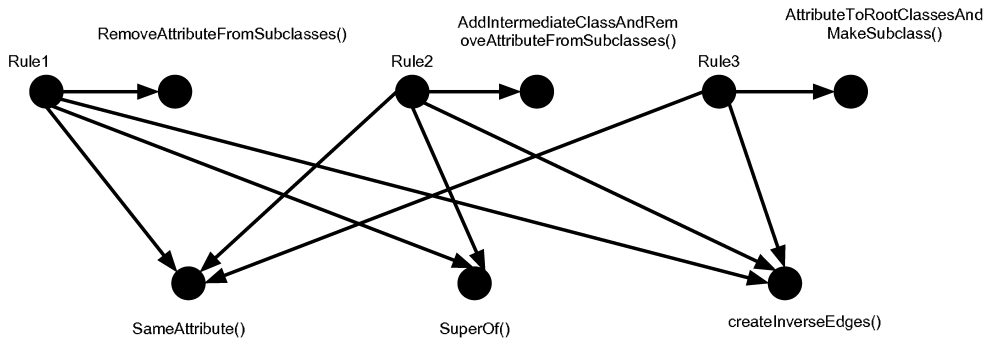

Fig. 9. Call graph of GrGen solution.

```
10    }
11    exec(AttributeToRootClassesAndMakeSubclass(a3, c) ;> [createInverseEdges]);
12    }
13 }
```

This definition relies upon the following helper pattern and rule:

```
1 pattern IsRoot(root: Class) {
2   negative {
3     . −:specific−> root;
4   }
5 }
6 rule AttributeToRootClassesAndMakeSubclass(attr:Property, super:Class) {
7   iterated {
8   :IsRoot(root); root:Class;
9   root −:ownedAttribute−> a:Property;
10  super; // super <> root
11  :SameAttribute(a,attr);
12  modify {
13    super −:specialisation−> gNew:Generalization −:specific−> root;
14    delete(a);
15  }
16 }
17 }
```

Although the *rule*3 rule appears to operate only on pairs of root classes, which would be highly non-optimal in situations such as test case 5, the auxiliary rule actually searches for all root classes with a given attribute and makes all of these subclasses of the class *c* created in *rule*3. Thus, the result for test case 5 is optimal: a single new root class is created by this rule, and one common attribute moved up to it, then the other shared attributes are moved up to this class by applications of *rule*1.

6.2. Evaluation properties

Table 9 summarises the evaluated attributes of the GrGen solution.

Although the GrGen solution is larger in size and structural complexity than the UML-RSDS solution, it exhibits lower overall complexity. The approach has higher maturity than UML-RSDS.

6.2.1. Transformation language properties

GrGen specifies transformations in terms of graph patterns, and hence its syntax is related to a visual presentation of graph nodes and edges, representing the abstract syntax of the source and target languages. In this respect it is at a high level of abstraction, although some imperative code fragments are required for complex transformations. The lack of universal quantifiers in source patterns results in an unclear formulation in terms of double negations, which makes the rules further from a direct logical interpretation, compared to UML-RSDS. The notation is semantically close to OCL, but uses a variant syntax.

The transformation uses a similar strategy as the UML-RSDS transformation, and does not guarantee an optimal restructuring in all cases, because of non-determinism in the order of matching model elements.

There is no static GrGen verification support for proving syntactic correctness. Nonetheless, it is simple to write GrGen assertions for the *Asm* constraints of the problem, and then to evaluate these in the target model to verify syntactic correctness. Inverse links to associations are set explicitly, unlike the implicit updating of inverse links used in UML-RSDS.

Fig. 9 shows part of the call graph of the GrGen.NET solution (there are additional calls to *SameAttribute* from *RemoveAttributeFromSubclasses*, etc., and an additional *isRoot* rule).

If the entire transformation is considered as a single module, then it has only internal calls, and cohesion is 100% and coupling 0%. If the rules were separated into different modules, one for each rule, then the helper patterns *SameAttribute*,

Table 12
Test case results for GrGen.

Test case	Number of classes	Number of attributes	Execution times (ms)	Effectiveness
1	5	5	80	100% (non-optimal)
2	7	7	100	100% (optimal)
3	20	20	90	100% (optimal)
4	105	132	100	100% (optimal)
5	500	5000	472 580	100% (optimal)

Table 13
Maximum capability test results for GrGen.

Test case	Number of classes	Number of attributes	Execution times (ms)	Effectiveness
$2 * 100$	400	400	1192 ms	100%
$2 * 200$	800	800	8332 ms	100%
$2 * 1000$	4000	4000	123 s	100%
$2 * 10\,000$	40 000	40 000	464 s	100%

SuperOf, *createInverseEdges* should be factored into an auxiliary module accessed by the rule modules. In this case the cohesion would be 28%. There are 17 graph pattern expressions of size 7 or more, of which 6 are non-unique, so the percentage of such unique expressions is 65%.

The transformation can be internally composed into a larger transformation process by means of a *xgrs* script sequentially composing the rules of the transformations, and externally composed by sequencing the generated C# implementations of transformations. Development effort was low.

GrGen.NET has been publicly available since 2003 and about 50 transformation case studies have been implemented using it.

In the usability survey, 4 of 5 participants considered the specification clearly structured, and with low or medium effort to understand. All respondents correctly identified the code used in the learnability test (no participant had prior knowledge of GrGen).

The transformation could be extended to the full UML 2 class diagram language, by means of additional rules and more detailed rule code. The factoring of *SameAttribute* into a separate helper pattern potentially reduces the effort of such extension.

6.2.2. Transformation implementation properties

A C# program is generated by GrGen.NET to implement the transformation. As with UML-RSDS this makes for an efficient implementation with a high maximum capability. The transformation engineer should never need to inspect this code, instead all work on the transformation should be performed at the specification level.

GrGen provides a convenient visual debugger which can be used for validation of a transformation. Faults arising from conflicting rules can be avoided by explicit scheduling of the rules, so that only one rule is applicable at a given time. Assumptions of a transformation can be checked in the source model. There are no runtime exception handling facilities embedded in synthesised transformations.

Table 12 gives the test results for GrGen, evaluated on the SHARE environment.

The maximum capacity results are shown in Table 13. The results for these were optimal.

7. Kermeta solution

Kermeta is a procedural language, built upon the type system of MOF [11]. Transformations are defined as sets of classes and operations which access and manipulate the elements of a model. Kermeta provides a meta-programming environment, with its own action language. Models may be manipulated by the creation and deletion of elements, and by the setting of element features. Logical data structures such as sets can be used to store selected sets of elements within a Kermeta program, and OCL-style operators such as *select* and *each* can be used to iterate over elements within a collection. Standard program control structures such as conditionals, sequencing and operation invocation can also be used. This 'programming with models' paradigm permits detailed control over the transformation processing.

7.1. Case study specification in Kermeta

The approach taken to solve the case study in Kermeta is to traverse through the model starting with the leaf nodes, and working upwards towards the root. To extract the first base set of nodes, which are the leaves, all the class nodes which have no children are selected (line 5 in the following listing). The lambda expression (lines 3 to 7) simplifies the code, enabling all the leaf classes to be extracted in a single line of code. This provides a set *leafNodes* of class nodes, which we can consider as a base set of classes to be processed by the rules.

```
1 var leafNodes : Set<String> init Set<String>.new
2 root.Eelement.select{ e | e.isInstanceOf(Class) }.each{ c |
3         var cl : Class
4         cl ?= c
5         if cl.specialisation.size() == 0  then
6                 leafNodes.add(cl.name)
7         end
8 }
```

The first part of the process when applying the rules is to ensure that the nodes for the next iteration are extracted. This is an important step and needs to be completed first, as the links below the parents could potentially be changed during the application of the given rule. Furthermore, each parent class should be added only once to the list, as duplicates in rules can cause considerable errors. In code segment 2 below, the code specifies the correct processing order, and note also that the list of classes are merely class names (line 15), this avoids the need to continuously clone classes. The parent nodes for each of the nodes in the base node are processed by calling the given rule, and on completion the parent node is marked as complete (line 26). Marking the node as processed, implies that all nodes below such a node have also been processed, and the next iteration should focus on the nodes above it.

```
1 var procNodes : Set<String> init Set<String>.new
2 var r1 : Rule1 init Rule1.new
3 r1.root := model.clone(root)
4 nodes.each{ cs |
5         var c : Class
6         c := r1.root.getClassFromString(cs)
7         c.specialisation.each{ p |
8                 var fd :  Boolean init true
9                 procNodes.each{ pc |
10                        if p.specific.name == pc and p.specific.done1 == void then
11                                fd := false
12                        end
13                }
14                if fd then
15                        procNodes.add(p.specific.name)
16                end
17        }
18 }
19 nodes.each{ cs |
20        var c : Class
21        c := r1.root.getClassFromString(cs)
22        c.generalisation.each{ g |
23                var cl : Class
24                cl := g.general
25                r1.ProcessSuper(cl)
26                cl.done1 := 1
27        }
28 }
```

For rule 1 “Pull up attributes” the Kermeta support for sets was used to extract the common set of properties. A set of properties is created and initialised with the owned attributes of one of the classes from the list which is marked as processed, then the intersection between this set and the set of properties, of the next class that is not yet processed, is calculated. The resulting set provides the common properties between the two classes, continuing in this manner the final set (*iset* on line 10 of the following code segment) should contain the set of properties which are common to all the classes. Also, at any time during the processing if the base set becomes empty the process can be terminated (line 16 in the following segment), implying that there are no common properties. The common properties are removed from the list, and added to the parent node (lines 14 and 13, respectively). A new iteration starts, by making use of the parents in the previous iteration as the base set of classes, and finally it terminates when there are no more classes in the base set.

```
1 var propSet : Set< Set<Property> > init Set< Set<Property> >.new
2 c.specialisation.each{ s |
3         var sc : Class
4         sc := s.specific
5         var prop : Set<Property> init Set<Property>.new
6         sc.ownedAttribute.each{ p | prop.add(p) }
7         propSet.add(prop)
8 }
9 var iset : Set<Property>
10 iset := Process(propSet)
11 if iset.size() > 0 then
12        stdio.writeln("Common Properties")
13        addPropertiesSuper(c,iset)
14        remPropertiesChildren(c,iset)
15 else
16        stdio.writeln("No Common Properties")
17 end
```

The approach taken to apply rule 2 “Create subclass”, follows a similar approach to walk the model as rule 1, the difference is the way in which the base set of nodes are processed. In this scenario, the set of classes extracted in each application of the rule are the largest set of subclasses which have common properties (line 17 below). *clsMgr* maintains,

for each child class *cl*, the set of other child classes which have some common attribute with *cl* (lines 4 to 14). *cont* is then a set of maximal size from these sets. This set is then processed and a new parent class is created and the properties are moved into the parent class (line 19). The classes which are not included in the largest set also need to be processed in order to completely ensure that the rule application is complete. In this case the remaining classes are grouped together and rule 2 is applied to these classes (line 20).

```
 1 stdio.writeln("Processing General: " + cl.name)
 2 var children : Set<Class> init cl.getChildren()
 3 var clsMgr : ClassMgr init ClassMgr.new
 4 children.each{ c1 |
 5         clsMgr.newSet(c1)
 6         children.each{ c2 |
 7                 if c1.name != c2.name then
 8                         if c1.checkCommon(c2) then
 9                                 stdio.writeln("Common found")
10                                 clsMgr.addClass(c1,c2)
11                         end
12                 end
13         }
14 }
15 stdio.writeln("Processing Containers")
16 var cont : Container
17 cont := clsMgr.getLargestSet()
18 if  cont.data.size() > 0 then
19         processProperties(cont,cl)
20         processRemProperties(cont,cl)
21 end
```

On test case 1, for example, the larger set $\{B, C, D\}$ of subclasses with $b : T2$ as a common attribute is chosen as *cont*, instead of the smaller set $\{A, B\}$ with $a : T1$ in common.

To apply rule 3 “Create root class”, we used a slightly different process as there is no need to walk through the model, all that needs to be done is to find all the root nodes in the structure and apply rule 1. In the case that there are common properties, a new root class is created (line 6 below). Using this root class the process is then to apply a similar process as for rule 2 to optimise the properties. In the case where there are not common properties, once again we apply rules similar to rule 2; however, now the process is applied to a list of classes as there is no common parent class.

```
 1 var r1 : Rule1 init Rule1.new
 2 r1.root := model.clone(root)
 3 if r1.ProcessClasses(nodes).size() > 0 then
 4         var name : String init ""
 5         nodes.each{ p | name.append(p) }
 6         var newClass : Class init Class.new
 7         newClass.name := name
 8         nodes.each{ cs |
 9                 var c : Class
10                 c := root.getClassFromString(cs)
11                 var spec : Generalization init Generalization.new
12                 spec.general := newClass
13                 spec.specific := c
14                 root.ERelation.add(spec)
15                 c.generalisation.add(spec)
16         }
17         r1.ProcessSuper(newClass)
18         root.Eelement.add(newClass)
19         root := model.clone(r1.root)
20 end
21 var r3 : Rule3 init Rule3.new
22 r3.root := model.clone(root)
23 r3.ProcessClass(nodes)
24 root := model.clone(r3.root)
```

The full listings of this solution can be found on the SHARE site for the paper [24].

7.2. Evaluation properties

Table 9 summarises the evaluated attributes of the transformation language and tools for the Kermeta solution.

Although the complexity and size of the Kermeta solution are significantly higher than the UML-RSDS and GrGen solutions, the language does provide more precise control over rule applications, for example, to take the largest collection of subclasses of a class which satisfy rule 2 as the first match for this rule. Such control is very complex or impossible to express in the more declarative languages.

7.2.1. Transformation language properties

Kermeta is at a low level of abstraction, and the solution is written in an exclusively imperative manner. This has negative consequences for the size and complexity of the solution, and also leads to a higher development effort; however, greater power is available to the specifier, and more detailed control over the transformation processing. This results in higher effectiveness than the declarative solutions, and imposes a specific deterministic processing to achieve confluence.

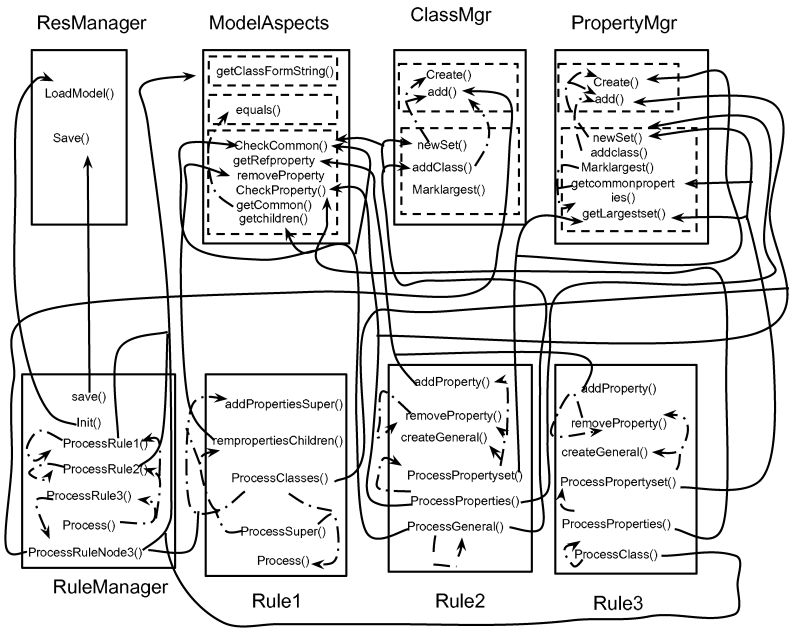

Fig. 10. Calling dependencies of Kermeta solution.

Table 14
Test case results for Kermeta.

Test case	Number of classes	Number of attributes	Execution times (ms)	Effectiveness
1	5	5	0	100% (optimal)
2	7	7	10	100% (optimal)
3	20	20	10	100% (optimal)
4	105	132	20	100% (optimal)
5	500	5000	out of memory	–

The transformation is decomposed into several modules and operations. Extension to a larger subset of UML 2 should be facilitated by this decomposition. However, a substantial amount of new transformation code would be required.

Fig. 10 shows the calling dependencies between the classes and operations of this solution. Dashed lines indicate internal dependencies (cohesion) of modules, solid lines indicate external dependencies (coupling). 56% of the calls are internal to modules. It can be seen that *ProcessRule1* and *ProcessRule2* are recursively defined. There are 17 expressions of size 7 or more, of which 9 are non-unique, giving a factorisation value of 53%.

In the usability survey, 3 of 5 respondants considered that there was no clear structure to the specification, 4 of 5 respondants found that a large or very large effort was required to understand it. 3 of 5 respondents correctly identified the code segment for the learnability test.

7.2.2. Transformation implementation properties

The use of recursion for the top-level control of the application of the rules makes proof of termination less clear than for UML-RSDS and GrGen. The fact that updates to the model are carried out in many places in the code also hinders proof of syntactic correctness. Kermeta provides an assertion capability, so that assumptions can be checked at runtime.

Table 14 shows the results of the four main test cases using Kermeta on the SHARE environment.

Table 15 gives the results of the maximum capability tests which could be executed for Kermeta on the SHARE environment. The construction of large sets of elements in the applications of rule 3 “create root class” appears to be the main source of execution costs and memory resource usage. Compared to the other solutions, more memory is consumed in the processing of sets of elements due to the overheads involved in enforcing a precise iteration order over the class hierarchy of a model.

Application of rule 3 was generally the most expensive component of the execution.

Kermeta has good support for interconnection with Eclipse. It has been available since 2005, and it has an extensive history of use, with over 40 published case studies (e.g., [35,33]).

Table 15
Maximum capability test results for Kermeta.

Test case	Number of classes	Number of attributes	Execution times (ms)	Effectiveness
$2*100$	400	400	13 s	100% (optimal)
$2*200$	800	800	100 s	100% (optimal)
$2*500$	2000	2000	1500 s	100% (optimal)
$2*1000$	4000	4000	12 600 s	100% (optimal)

8. ATL solution

ATL [21] is a hybrid model transformation domain-specific language containing a mixture of declarative and imperative constructs. ATL transformations are unidirectional, operating on read-only source models and producing write-only target models. During the execution of a transformation, source models may be navigated but changes are not allowed. Target models cannot be navigated.

ATL modules define the transformations. A module contains a mandatory header section, an import section, and a number of helpers and transformation rules. The header section provides the name of the transformation module and declares the source and target models (which are typed by their metamodels). Helpers and rules are the constructs used to specify the transformation functionality.

Declarative ATL rules can be classified as matched rules and lazy rules. Lazy rules are like matched rules, but are only applied when called by another rule. They both specify relations between source patterns and target patterns. The source pattern of a rule specifies a set of source types and an optional guard given as a Boolean expression in OCL. A source pattern is evaluated to a set of matches in source models. The target pattern is composed of a set of elements. Each of these elements specifies a target type from the target metamodel and a set of bindings. A binding refers to a feature of the type (i.e., an attribute, a reference or an association end) and specifies an expression whose value is used to initialise the feature. Lazy rules can be called several times using a collect construct. Unique lazy rules are a special kind of lazy rules that always return the same target element for a given source element. The target element is retrieved by navigating the internal traceability links, as in normal rules. Non-unique lazy rules do not navigate the traceability links, but create new target elements in each execution.

In some cases, complex transformation algorithms may be required, and it may be difficult to specify them in a declarative way. For this reason, ATL provides two imperative constructs: called rules and action blocks. A called rule is a rule called by other ones in a procedural style. An action block is a sequence of imperative statements and can be used instead of, or in combination with, a target pattern in matched or called rules. The imperative statements in ATL are the usual constructs for attribute assignment and control flow: conditions and loops.

ATL has two execution modes, the normal (default) execution mode and the refining one:

- In the default execution mode, the ATL developer has to specify the way to generate each of the expected target model elements. This execution mode suits most ATL transformations where source and target models are different.
- The refining execution mode was introduced to ease the programming of in-place transformations. With the refining mode, ATL developers can focus on the ATL code dedicated to the generation of modified target elements.

Ideally, the transformation in our case study should be defined using the refining mode, since we are dealing with an in-place transformation. However, the current version of the refining mode still faces many limitations, as we describe in Appendix A. For this reason, we present both a solution given in an ideal and hypothetical ATL refining mode (given in Appendix A), and another one in the following section in the default mode. Although we only define one rule of the original specification from Section 3 in this mode, this definition is representative of the style of ATL default mode specification which can be used for the case study, and the other rules can also be defined using the same approach. A more imperative variation of the solution could be given, however this would not overcome the essential limitation of the default mode, i.e., that update-in-place transformations can only be simulated by selective copying.

8.1. Case study specification in ATL

This implementation (for rule 1 of the case study) is composed of 5 matched rules, 1 lazy rule, and a variable which is used internally in a rule. The reason that there are so many rules is that we need a matched rule to copy objects of each type in the metamodel. This is because in the default mode, those objects not explicitly copied by rules do not appear in the target model.

The two simplest matched rules are the following. The first one copies objects of type *Generalization*, while the second one copies objects of type *Type*.

```
1 rule Copygeneralization{
2     from
3         g : ClassDiagram!Generalization
4     to
```

```
5         gOut : ClassDiagram!Generalization(
6             specific <- g.specific,
7             general <- g.general
8         )
9 }
10
11 rule CopyType{
12     from
13         t : ClassDiagram!Type
14     to
15         tOut : ClassDiagram!Type(
16             name <- t.name
17 )
18 }
```

When applied to test case 2 from Section 4.5, these rules copy the four generalisations and two types ($T1$ and $T2$) from the source to the target model.

We need to be careful when copying the objects of type *Property*. We differentiate between those properties contained in classes which are subclasses, (i.e., those classes which have generalizations) and those contained in classes which are not subclasses (those classes which do not have any generalizations). The following matched rule copies all the properties contained in classes which are not subclasses.

```
1 -- This rule copies properties (all of them) from classes which are not
2 -- subclasses.
3 rule CopyPropertiesNonSubclasses{
4     from
5         p : ClassDiagram!Property,
6         c : ClassDiagram!Class(c.ownedAttribute->includes(p) and
7                                c.generalisation->size() = 0)
8     to
9         pOut : ClassDiagram!Property(
10            name <- p.name,
11            type <- p.type
12        )
13 }
```

The matching in the rule's LHS is performed with two objects: the property to be copied and the class that contains it. It is checked that the class is not a subclass. Applied to test case 2, the rule copies the attributes $A::a$, $D::a$ and $G::b$.

For other classes, the properties to be copied are those which do not appear in all the siblings of their direct superclass. The following rule deals with this case.

```
1 -- This rule copies those properties belonging to subclasses which are
2 -- not repeated in all the subclasses of a class.
3 rule CopyPropertiesSubclasses{
4     from
5         p : ClassDiagram!Property,
6         c : ClassDiagram!Class,
7         cSup : ClassDiagram!Class(c.ownedAttribute->includes(p) and
8                c.generalisation->exists(g | g.general = cSup) and
9                cSup.specialisation->exists(g | not
10               g.specific.ownedAttribute->exists (pr | pr.name = p.name)))
11    to
12        pOut : ClassDiagram!Property(
13            name <- p.name,
14            type <- p.type
15        )
16 }
```

The matching in this rule is performed with three objects: the property p to copy, the class c that contains it (which is a subclass) and the superclass *cSup* of such a class. In the condition in the rule's LHS, it is checked that there is a sibling of the subclass which does not contain a property with the same name as the one in the matching. This rule is not applicable to any elements in test case 2.

Finally, there is a matched rule to copy all the classes. Special care has to be taken when copying the elements in the *ownedAttribute* reference, because those properties which are repeated in all the subclasses of a class must not be copied. Furthermore, if the class being copied is a superclass all of whose subclasses contain a property with the same name and type, a property with such name and type should be created and referenced by the class created. The rule is the following.

```
1 -- This rule copies all the classes and adds to those new
2 -- classes those attributes created by rules CopyPropertiesSubclasses
3 -- and CopyPropertiesNonSubclasses, when it corresponds. Besides,
4 -- it adds new attributes in a superclass when it corresponds.
5 rule CopyClasses{
6     from
7         c : ClassDiagram!Class
8     to
9         cOut : ClassDiagram!Class(
10            name <- c.name,
11            generalisation <- c.generalisation,
12            specialisation <- c.specialisation
```

```
13         )
14 do{
15    -- If c is not a subclass, all the attributes are copied by
16    -- rule CopyPropertiesNonSubClasses, so we make the new
17    -- class contain them
18    if (c.generalisation -> size() = 0){
19        cOut.ownedAttribute <- c.ownedAttribute;
20    }else{
21    -- If c is a subclass, those attributes not repeated in its
22    -- siblings are copied by rule CopyPropertiesSubclasses,
23    -- so we add them here.
24       -- We assume there is not multiple inheritance
25       thisModule.cSup <- c.generalisation->first().general;
26       for (p in c.ownedAttribute){
27           if (thisModule.cSup.specialisation->exists(g |
28           not g.specific.ownedAttribute->exists( pr | pr.name
29           = p.name))){
30               cOut.ownedAttribute <- cOut.ownedAttribute->append(p);
31           }
32       }
33    }
34    -- Furthermore, if c is a superclass, we create a new attribute in
35    -- it if all its subclasses contain an attribute with the same
36    -- name and type
37    if (c.specialisation->size() >0){
38        for (p in c.specialisation->first().specific.ownedAttribute){
39            if (c.specialisation->forAll(g | g.specific.ownedAttribute
40            ->exists(pr | pr.name = p.name and pr.type = p.type))){
41               -- A call to lazy rule CreateProperty is made
42               cOut.ownedAttribute <- cOut.ownedAttribute->
43               append(thisModule.CreateProperty(p));
44            }
45        }
46    }
47 }
48 }
```

The copy of attributes is realised by the rule's imperative part. If the class in the rule's LHS is not a subclass, then all their attributes are copied (using the ← operator on line 19 of the above code segment). ATL's engine looks in the internal traces for those properties created by rule *CopyPropertiesNonSubClasses* and makes the new class reference them. If the class is a subclass, the rule copies those attributes which are not repeated in all the siblings of the class. Finally, the last part in the imperative section checks, if the class is a superclass, that all of its children own attributes with a common name and type. In such cases, it creates a new property with the lazy rule *CreateProperty* and makes the class reference it. The lazy rule receives a property as argument and creates a new property with the same features. The lazy rule is:

```
1 -- lazy rule that copies the property received as argument
2 lazy rule CreateProperty{
3     from
4         p : ClassDiagram!Property
5     to
6         pOut : ClassDiagram!Property(
7             name <- p.name,
8             type <- p.type
9         )
10 }
```

Applied to test case 2, *CopyClasses* copies all the classes and links them by their copied generalisations. For classes A, D and G their copied attributes are assigned to the class copies. The last part of the rule and *CreateProperty* creates the $A::b$ and $D::b$ attributes in the target and adds them to the copies of A and D respectively.

8.2. Evaluation properties

The properties of the ATL default mode solution for rule 1 are shown in Table 9.

8.2.1. Transformation language properties

The declarative parts of the ATL language are at a high level of abstraction; however, this type of problem requires the use of imperative features, as in the solution for rule 1 presented here. Therefore we classify the solution as hybrid.

The ATL approach used for rule 1 would suffer potentially from the same problems of lack of confluence and failure to achieve 100% effectiveness as the UML-RSDS and GrGen.NET solutions, if used for the other rules: no control over the order of applications of a rule to elements is possible.

The specification is relatively similar to the informal description of the transformation; however, a significant difference is the separation of the copying of unchanged parts of the model (which is implicit in the rule descriptions) from the explicit modifications needed (promotion of the duplicate copies of an attribute). The latter is performed in the *CopyClasses* rule.

The need for explicit copying rules leads to a high syntactic complexity, as do the imperative code blocks in *CopyClasses*. Structural complexity is 8 since there are 6 invocations of rules by other rules within the specification (5 implicit calls and 1 explicit), and a maximum call depth of 2. Fig. 11 shows the call graph.

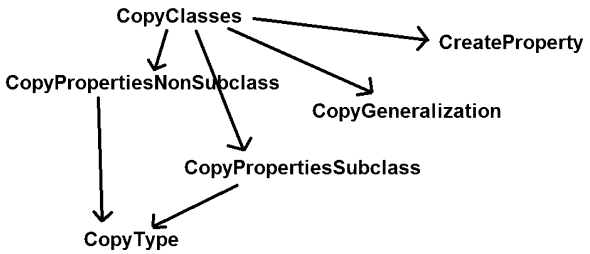

Fig. 11. Call graph of ATL solution (rule 1).

Table 16
Test case results for ATL.

Test case	Number of classes	Number of attributes	Execution times (ms)	Effectiveness
2	7	7	84	40% (non-optimal)

Table 17
Maximum capability test results for ATL.

Test case	Number of classes	Number of attributes	Execution times (ms)	Effectiveness
$2 * 100$	400	400	1142 s	40%
$2 * 200$	800	800	10 888 s	40%
$2 * 500$	2000	2000	150 999 s	40%
$2 * 1000$	4000	4000	173 292 s	40%
$2 * 5000$	20 000	20 000	147 978 s	40%

As with the UML-RSDS solution it is possible to reason about general properties, for example, that the set of classes is not diminished by the transformation. Proof of syntactic correctness is complicated by the copying strategy, e.g., it is not clear if *CopyClasses* can introduce duplicated attributes of a class. There is no simple variant which is decreased by each rule application, unlike the solutions of UML-RSDS and GrGen. This makes formal proof of termination more difficult.

There is no formal tool support for verification. Some initial work on ATL verification using SMT provers has been carried out, for simple declarative ATL transformations in default mode [8].

One ATL transformation can be sequentially composed with another (the transformations being defined in separate modules) by means of a Java script. This technique is needed here to carry out the fixed-point iteration of rule 1.

The use of standard OCL notation within rules assists in the interoperability and comprehensibility of ATL specifications. ATL is a relatively mature language. It was first available in 2003, and has over 100 published case studies [5].

There are 32 expressions of complexity 7 or above, of which 9 are duplicated, so the percentage of unique expressions of complexity 7 or above is 72%.

Extension to full UML class diagrams could be carried out in principle, with considerable additional complexity to the rules, and with additional rules (at least one copy rule for every entity type in the metamodel – there are over 30 such entity types). This effort may be eased for UML by using the existing UML2Copy.atl module for copying UML model data [42]. However, the rule 1 copy rules for generalisations and properties cannot be reused for rule 2 and 3 of the problem, since different copying criteria are necessary for these rules, and this leads to a multiplicity of variants of copying rules.

8.2.2. Transformation implementation properties

The above ATL solution defines one application of rule 1, and this clearly terminates, because the ATL rules are all linear iterations over their source domains. However, termination of the completed solution would be difficult to show, because the individual ATL rules do not reduce any simple variant value such as $Property.size$. Confluence may fail, since different orders of iteration over the source domains may produce different result models. ATL provides runtime checking for rule conflicts, and gives error messages indicating the specification lines responsible, if there is a runtime error.

As with the other solutions, the ATL solution depends on the absence of multiple inheritance in the source model in order to operate correctly.

Table 16 shows the test case results for ATL on the SHARE environment. Only test case 2 was attempted. All possible applications of rule 1 were performed successfully.

The maximum capability tests were carried out as shown in Table 17. Only applications of rule 1 were performed in each case. While the ATL tools were able to execute all of the capacity test cases, up to size 50 000 elements, only the first (with an execution time of 19 minutes) completed in a reasonable time.

9. QVT-R solution

In this section we describe the application of the QVT-Relational language and Medini QVT tools to the quality improvement case study. QVT-R is a declarative model-transformation language, defined in an OMG standard [36]. It is based on the concept of relations between the source model(s) and target model(s): the transformation attempts to establish a number of relations between source and target models by creating or modifying elements in the target models. Individual relations typically relate source elements of a specific source entity type to target elements of specific target entity types. A relation can rely on certain previous properties and relations already being established (the *when* clause of a relation), and can invoke subordinate relations and actions (the *where* clause). Transformation execution is initiated by *top relations*, which operate in non-deterministic order upon all matching elements in source models.

9.1. Case study specification in QVT-R

It is complex to specify this task in QVT-R because of the lack of default copy rules. Such rules can be defined explicitly but this is not trivial for cases where large parts of a model are copied and other parts are deleted or modified. We have adopted the strategy of using marker relations from [12] to implement this transformation. The elements to be modified are selected and marked in a first step, then the input model is copied to an updated target model, with specific rules applied to the marked elements, and deleted elements are not copied. We found this approach simpler than using an update-in-place strategy for the transformation.

9.1.1. Rule 1 "Pull up attributes"

This rule is implemented by identifying those *Generalization* objects such that the linked classes of the object satisfy *rule1*. In order to generate the list of such *Generalization*, we declare a top relation, which is applied to every instance of *Generalization*. The *when* clause of this rule declares that the *general* end (superclass) has more than one subclass, and more than one subclass has attributes with the same name and type. In the *where* clause of this rule another relation *CheckRule1* is called which has the *Generalization* instance as its argument.

```
transformation QualityImprovement(source:KCL, target:KCL)
{
  top relation TakeGeneralization {
    checkonly domain source sourcegen : KCL::Generalization {
      specific = sub : KCL::Class {
        ownedAttribute = prop : KCL::Property{}}};
    enforce domain target targetgen :KCL::Generalization {};
    when { sourcegen.general.specialisation.size() > 1 and
      sourcegen.general.specialisation->exists(
        c | c.specific.name <> sub.name and
          c.specific.ownedAttribute->exists(
            p | p.name = prop.name and p.type.name = prop.type.name) ); }
    where { CheckRule1(sourcegen,targetgen); }
}
```

Here, KCL is the name of the metamodel of the transformation (Fig. 1).

It is required to prioritise *Rule1* over *Rule2*. This can be ensured by using *CheckRule1*, which specifies that no sibling class exists without a copy of the property, i.e., that all of the subclasses have an attribute with the same name and type.

```
relation CheckRule1 {
  checkonly domain source sourcegen : KCL::Generalization {
    specific = subsource : KCL::Class {
      ownedAttribute = prop : KCL::Property{} } };
  enforce domain target targetgen :KCL::Generalization
    { specific = subtarget : KCL::Class {} };
  when { sourcegen.general.specialisation.size() > 1 and
    sourcegen.general.specialisation->exists(
      c | c.specific.name <> subsource.name
and
        not c.specific.ownedAttribute->exists(
          p | p.name <> prop.name) ); }
  where { CopyGeneralisationForRule1(sourcegen,targetgen) and
    CopyClass(subsource,subtarget); }
}
```

If the conditions of *rule1* are satisfied, the corresponding *Generalization* is copied. The *CopyGeneralisation* rule is a non-top relation and only matches its arguments in the source and target model [12]. It acts as a marker relation to identify those generalisations that meet the conditions of rule 1. Likewise for *CopyClass*.

```
relation CopyGeneralisationForRule1 {
  checkonly domain source sourcegen : KCL::Generalization {};
  enforce domain target targetgen : KCL::Generalization {};
}
```

The next step is to assign a superclass to the selected *Generalization*. The *TakesuperClass* relation takes all the classes from the input model and in the *when* clause of the relation selects the classes that have a *specialisation* element which has

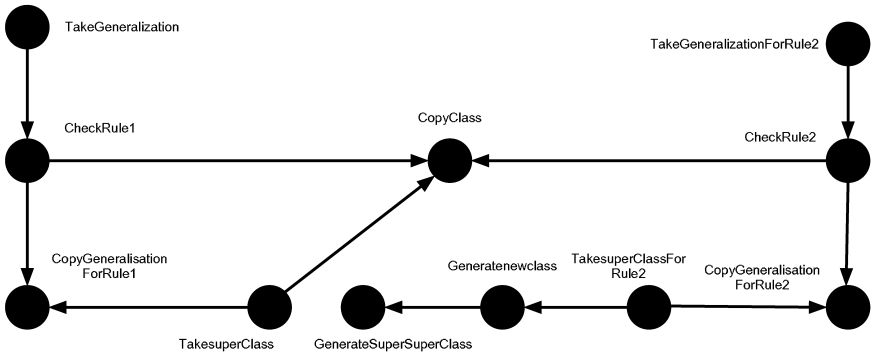

Fig. 12. Call graph of QVT-R solution.

already been copied by the *CopyGeneralisation* rule. Furthermore, this rule performs the main task of moving the property from the subclasses to the superclass. This happens by accessing the property of the subclass in the *checkonly domain* of the relation and then adding it into the owned properties of the superclass. In order to copy the name of the superclass the relation *CopyClass* is called in the *where* clause of this rule.

```
top relation TakesuperClass{
  checkonly domain source s : KCL::Class {
    specialisation = spsourcesuper : KCL::Generalization {
      specific = spe : KCL::Class {
        ownedAttribute = prop : KCL::Property {} } } };
  enforce domain target t :KCL::Class {
    specialisation = sptargetsuper : KCL::Generalization {},
    ownedAttribute = prop : KCL::Property {} };
  when { CopyGeneralisationForRule1(spsourcesuper,sptargetsuper); }
where { CopyClass(s,t); }
}
```

The strategy for Rule 2, “create subclass” is similar and can be found in Appendix B.

9.2. Evaluation properties

Table 9 summarises the evaluated attributes of the transformation language and tools for the QVT-R solution.

9.2.1. Transformation language properties

The QVT-R language is at a very high level of abstraction, and the solution is expressed in terms of logical relations. However, the nature of the problem, which involves copying, creating, deletion and modification of model elements, leads to an obscure specification of the transformation in QVT-R.

The transformation is of moderate size and complexity, and has been modularised by factorisation of conceptually distinct predicates, and by sequential decomposition, into a selection phase followed by a restructuring phase. Fig. 12 shows the call graph of this solution. There are 46 expressions of size 7 or more, of which 28 are unique, giving a factorisation of 61%.

If the transformation is modularised into two modules according to Fig. 12, then there is 1 inter-module call (from *CheckRule*2 to *CopyClass*) and 10 intra-module calls, i.e., the cohesion is 91%.

The correctness of the transformation cannot be established because of some remaining problems:

- The transformation removes copies of a property which has clones from subclasses but also it adds all copies into the superclass (it does not keep one copy and destroy the other copies). Therefore, we need additionally to remove duplicate copies from the superclass.
- It is not possible to prioritize the rules in QVT: even though the conditions for *rule1* and *rule2* have been made disjoint, both transformations could potentially apply at different locations in a model at the same time.
- The transformation is not complete. The QVT relational language does not support any technique for *rule3* “Create root class”, as there is no connection in the model between different root classes.
- Further copy rules are needed to map parts of the source model which are not matched by the restructuring rules.

For similar reasons it seems unlikely that the approach could be easily extended to larger class diagram metamodels.

Development effort for the transformation was quite high, and needed research into specialised patterns for defining copy transformations in QVT-R.

In the usability survey three respondants considered the structure to be clear, but all considered the understandability effort to be large or very large.

Table 18
Test case results for QVT-R.

Test case	Number of classes	Number of attributes	Execution times (ms)	Effectiveness
2	7	7	109	40% (non-optimal)
3	20	20	159	55% (non-optimal)
4	105	132	297	61% (non-optimal)
5	500	5000	Out of memory	–

Table 19
Maximum capability test results for QVT-R.

Test case	Number of classes	Number of attributes	Execution times (ms)	Effectiveness
$2 * 100$	400	400	1610 ms	40%
$2 * 200$	800	800	2359 ms	40%
$2 * 500$	2000	2000	38 s	40%

9.2.2. Transformation implementation properties

As with the other declarative and hybrid language solutions, the confluence of the QVT-R solution cannot be established. Termination should follow since no recursion is used; however, the two-phased copying approach means that there is no simple variant which is reduced by every rule application.

The Medini tool provides run-time exception handling: errors in processing result in termination of the transformation and an error message giving the specification lines where the error occurred.

For test case 1 there should be one application of rule 2. However, in the implementation all a and b properties are moved to the same new superclass. Test case 2: the implementation performs this testcase correctly for rule 1 (not rule 3) and the execution time is 109 ms. Test case 3 takes 159 ms but is not completely processed. Test case 4: the implementation performs all rule1 applications which can be done in the first iteration, removing 56 of 92 clones. The time taken is 297 ms. Test case 5 could not be completed (out of memory error).

Table 18 shows the test case results for QVT-R on the SHARE environment.

The maximum capability tests were carried out as shown in Table 19. Processing of larger models was not possible (an out of memory error was produced). Only applications of rule 1 were performed.

QVT-R is defined by an international standard [36]. Version 1.0 was released in 2008, and there are 20 published case studies. Medini QVT has been available since 2007.

10. Comparison of approaches

In this section we evaluate the relative capabilities of each transformation approach, and we provide a summary ranking of the approaches according to the ISO 9126-1 quality characteristics.

10.1. Comparative evaluation of quality measures

We give comparisons of each approach with respect to the quantitative metrics of size, complexity, modularity and development effort. We also summarise the results of the usability survey, and discuss what correlations exist between different measures for the solutions.

10.1.1. Size

A simple size metric is the number of lines of code in a transformation specification. Line counts have deficiencies as measures, but they provide a quick comparison of the relative size of different transformation specifications. Table 9 shows the size measures for the solutions.

The declarative approaches have generally lower size metrics than the hybrid or imperative approaches. As in the following sections, it should be noted that the ATL and QVT solutions only covered one and two of the three rules of the problem, respectively. If these solutions were scaled up by a factor of 3 and 1.5, respectively, then a complete ATL solution would have an estimated size of 243 lines, and a complete QVT-R solution 125 lines. This does not change the classification of these solutions as Medium size.

The ranking of the solutions on the basis of size (using the scaled-up values for QVT-R and ATL) is: UML-RSDS; GrGen; QVT-R; ATL; Kermeta.

Size correlates inversely with abstraction level, except that QVT-R requires a larger specification than its abstraction level would suggest, due to the limitations of the language for this category of problem.

10.1.2. Complexity

Table 9 compares the complexities of the approaches, based on the syntactic complexities, the structural complexities and the overall complexities.

Table 20
Development effort.

	Rule 1 (min)	Rule 2 (min)	Rule 3 (min)	Summary
UML-RSDS	40	30	50	Medium
GrGen	40	30	30	Low
Kermeta	120	160	160	High
ATL	150	–	–	Medium
QVT-R	120	160	–	Medium

Again, Kermeta has significantly higher complexity than the more declarative approaches. ATL (scaled) is however in second place, due to the complex processing needed for this problem. UML-RSDS, whilst very concise in terms of lines of code, has in compensation a high density of complex expressions within its constraints.

Scaling up the complexities of the ATL and QVT-R solutions produces the figures of 456 and 285 respectively, which does not alter their classification on the three-point scale. The ranking of the complete solutions on the basis of complexity is: GrGen; UML-RSDS; QVT-R; ATL; Kermeta.

The Pearson correlation coefficient of Size with Complexity (using all five solutions, with scaling of ATL and QVT-R) is 0.99.

10.1.3. Development effort

The development effort of a transformation refers to the time spent on expressing each individual rule of the requirements in the specific transformation language (the time for understanding the problem or the specification notation is not considered here, nor is the time taken in constructing test cases, but iterative testing during writing the specification is included).

Table 20 shows the development effort time for each rule in the five different transformation languages. Table 9 presents a summarised view of the same data.

Unsurprisingly, the larger and more complex the specification, the more effort was required to develop it, with Kermeta and ATL (scaled) again the highest in this score. Notice however that the effort per line is actually lower for Kermeta than for UML-RSDS.

If the effort for ATL and QVT-R are scaled up, they have values of 450 and 420 respectively, placing them close to Kermeta in terms of effort. The ranking of the complete solutions on the basis of development effort is: GrGen; UML-RSDS; QVT-R; Kermeta; ATL.

The correlation of size with development effort, using all the solutions, with scaling, is 0.58. The correlation of complexity with development effort, using all the solutions, with scaling, is 0.67.

10.1.4. Correctness

Correctness was assessed by three separate 5-point measures of syntactic correctness (ability of the approach to prove that the target models constructed satisfy the language constraints *Asm*), termination (ability of the approach to prove termination) and confluence (ability of the approach to prove confluence or determinacy of the transformation).

Table 9 shows the individual results for each of these aspects, and an overall correctness value. The overall correctness value is based on the average of the values for the three specific correctness measures.

UML-RSDS supports proof of syntactic correctness by direct induction over transformation steps that *Asm* is preserved. Inverse links of associations are set automatically. Termination is proved by showing that *Property.size* is strictly decreased by each transformation step. However, the lack of detailed control over the transformation execution means that confluence cannot be proved (and indeed it fails – simply re-ordering the elements in the input model file can produce non-isomorphic result models). For GrGen the results are similar, except that syntactic correctness is less clear because the specification notation is further removed from pure logic, with operation calls and sequencing in the rule definitions. Unlike UML-RSDS, there is no tool support for correctness proof. Again, confluence fails. Kermeta is able to enforce an optimal clone-removal strategy and achieve confluence in principle. Establishing syntactic correctness and termination is obstructed by the complex control flow and use of recursion. The QVT-R solution has minor errors in syntactic correctness. For both ATL and QVT-R confluence fails, and proof of termination is obstructed because there is no simple variant that is reduced by each rule application. The ordering of the solutions with regard to correctness is: Kermeta; UML-RSDS; GrGen; ATL; QVT-R.

10.1.5. Usability factors

Understandability, learnability and attractiveness of the approaches (restricted here to these aspects as they concern the specification language, rather than the tool) were evaluated by a survey of informatics professionals ranging from PhD students to senior academics.

Table 21 summarises the results of the survey which was described in Section 4.2.3. The answers are coded on a five-point scale from 0 to 4. For each question we consider the average value of the response across the different respondants. The table also includes derived values, such as *Overall Understandability*, calculated from the perceived understandability *U1* and the actual understanding *U2*.

Table 21
Summary of usability survey question responses.

Question	*UML-RSDS*	*GrGen*	*Kermeta*	*ATL*	*QVT-R*
Expertise level *(E)*	0.4	0.4	0.4	1.0	0.8
Perceived Understandability *(U1)*	1.2	2.2	0.6	1.8	0.4
Actual Understanding *(U2)*	1.6	2.4	3.2	2.6	1.0
Overall Understandability	***1.4***	***2.3***	***1.9***	***2.2***	***0.7***
(= AVG(U1, U2))	Low	Medium	Medium	Medium	Low
Perceived Learnability *(L1)*	1.4	1.8	0.8	1.8	0.4
Actually Learned *(L2 = U2 − E)*	1.2	2.0	2.8	1.6	0.2
Overall Learnability	***1.3***	***1.9***	***1.8***	***1.7***	***0.3***
(= AVG(L1, L2))	Low	Medium	Medium	Medium	None
Well structured solution *(A1)*	1.6	2.8	1.2	2.8	1.2
Attractive Notation *(A2)*	2.8	1.2	0.6	2.0	2.4
Overall Attractiveness	***2.2***	***2***	***0.9***	***2.4***	***1.8***
(= AVG(A1, A2))	Medium	Medium	Low	Medium	Medium

Table 22
Correlation results of measures.

	Size	Complexity	Dev. effort	Usability	Modularity
Size		0.99	0.58	−0.55	−0.88
Complexity	0.99		0.67	−0.66	−0.90
Dev. Effort	0.58	0.67		−0.68	−0.72
Usability	−0.55	−0.66	−0.68		0.69
Modularity	−0.88	−0.90	−0.72	0.69	

The overall ranking of approaches with regard to usability is therefore: ATL (6.3); GrGen (6.2); UML-RSDS (4.9); Kermeta (4.6); QVT-R (2.8).

This ordering is inversely correlated with the values for complexity and size and development effort, and positively correlated with modularity, as could be expected by analogy with similar results for programs [9].

The Pearson correlation coefficient of usability with complexity for Kermeta, UML-RSDS and GrGen is −0.663, indicating a moderately strong linear dependency. The correlation coefficient of usability with size for Kermeta, UML-RSDS and GrGen is −0.55. The correlation coefficient of usability with modularity is 0.69 for Kermeta, UML-RSDS, GrGen and ATL, indicating a positive linear correlation.

10.1.6. Modularity

As discussed in Section 4, modularity is considered both in terms of how factorised the specification is and how the proportion of calls within modules (cohesion) relates to the proportion of calls between modules (coupling). Table 9 includes one row for the overall modularity of the specifications, and one row for each contributing factor (factorisation, cohesion, coupling).

Only the Kermeta and QVT specifications were decomposed into modules on the basis of the transformation rules. Modules consisted of the main rule processing for each of the required rules, together with auxiliary rules/operations for this processing. All the notations are capable of increasing the factorisation to 100%, and of further modularising the problem, so the UML-RSDS, ATL and GrGen solutions also have a potential modularity score of High.

Modularity is negatively correlated with development effort: −0.72 when all the solutions are considered, using scaling of effort for ATL and QVT-R, and the numeric value of factorisation + cohesion for modularity. Likewise it is negatively correlated (−0.88) with size and (−0.90) with complexity.

The ordering of the solutions with respect to modularity is: ATL; UML-RSDS; GrGen; QVT-R; Kermeta.

10.1.7. Correlation results

Table 22 summarises the correlation results we have obtained between the different measures of the solutions. Except as indicated above, the correlations considered all five solutions, with measures of size, complexity and development effort for ATL and QVT-R being scaled up. The correlation results indicate that the expected relationships hold between these measures, and give some evidence of the validity and consistency of the selected measures.

10.2. Evaluation summaries

For each approach, we can generate a summary of its quality characteristics based upon the measured values for its attributes. The different attribute measures could be combined using some scheme of numeric weighting, and likewise for the combination of subcharacteristics which contribute to a quality characteristic. Due to space limitations we will use a simple count of the attribute values for each subcharacteristic which score +1 or more on the three or five-point scales for the attribute/subcharacteristic (in Table 9). This allows decision-makers to gain a quick overview of the merits of a particular

Table 23
Overall ranking of approaches.

Approach	Funct. (13)	Rel. (2)	Usab. (3)	Effic. (2)	Maint. (3)	Port. (1)	Total (24)
UML-RSDS	8	0	0	2	1	0	11
GrGen	8	1	0	1	1	0	11
Kermeta	6	1	0	0	0	0	7
ATL	3	2	0	0	1	0	6
QVT-R	3	1	0	0	0	0	4

approach, in the event of a close contest between two approaches, the detailed measurement values for the approaches can be compared. The values of this count are not affected by scaling up the size, complexity and development effort for ATL and QVT-R.

In conclusion, we could consider GrGen, UML-RSDS and Kermeta as good candidates to select for carrying out this transformation, on the basis of their overall rankings in terms of the ISO 9126-1 characteristics (Table 23).

It should be noted that this ranking of approaches is specific to this type of refactoring problem: in particular, GrGen and UML-RSDS are favoured over ATL and QVT-R because of their stronger support for update-in-place transformations. The inefficiency of model copying solutions for such problems is evident in the execution results for ATL and QVT-R compared to the other solutions. Different categories of transformation problem, such as model migrations, would produce different rankings.

It is notable also that none of the approaches satisfies more than half of the quality subcharacteristics, which is perhaps due to the relative immaturity of the model transformation field. In particular, none of the approaches satisfy any usability characteristics.

The fundamental problems of this case study are:

- *The need to match multiple elements, including sets of arbitrary size.*
- *The need to control the order of application of rules.*
- *The need to support update-in-place transformation.* The transformation is input-destructive and it is very inefficient to implement by means of copying models.
- *The need to provide optimisations.* The complexity of expressions used in matching and updates would produce inefficient implementations without the use of optimisation.
- *The need to support fixpoint iteration of rules.*

The solutions have addressed these problems in different ways, depending on the facilities available in the different languages:

Multiple element matching: In the UML-RSDS solution this is addressed by using multiple quantifiers iterating over quantified variables which serve as pivot elements for the constraint, and deriving the sets of interest (sets of sibling classes with cloned attributes) from these multiple pivot elements.

In GrGen, the sets of elements are specified implicitly by means of graph patterns, positive and negative. Modifications involving such sets are performed by procedural iteration of updates applied to the individual elements.

In the Kermeta solution, sets of elements are constructed by iterations over the sets of all classes or attributes in the model, these sets are then passed to operations which update the model based on the sets.

In ATL multiple elements can be gathered into sets by means of the *using* construct, the sets are derived from the input elements of the rule.

In the QVT-R solution, the sets of classes containing copies of a specific attribute need to be precomputed by a sequentially preceding transformation. The attributes which have copies to be removed are identified by the 'marking' pattern of [12]. These steps considerably complicate the transformation specification and are a consequence of the element-by-element pattern matching and processing in QVT rules.

A more powerful solution than these would be to provide direct quantification/matching over sets of elements as pivots for transformation rules, as proposed for the CGT language in [15].

Control of rule ordering: In UML-RSDS rule applications are ordered based on rule priority: rule 1 takes precedence over rule 2, and rule 2 over rule 3. The priority is syntactically defined by rule ordering.

In GrGen, an explicit *xgrs* statement defines the processing order of the rules.

In Kermeta, the ordering of the rules is defined by an explicit algorithm invoking the rules in the required order.

In ATL, the ordering in matched rules cannot be controlled. Some matched rules may however depend on others, if they need elements created by the other rules. The ATL engine handles this by means of traceability links. Regarding declarative rules, matched rules do have preference over lazy (and unique lazy) rules, since the latter are executed when invoked from the former.

In QVT-R, the relative ordering of the top-level relation executions can be controlled by the use of *when* and *where* clauses, a rule that should occur after another should contain a *when* clause condition that can only be established by its predecessor. An example is the *CopyGeneralisationForRule1* relation in the above specification.

A fine level of control, e.g., to iterate over classes from the leaves of the inheritance hierarchy upwards, is required to give an optimal solution to the problem. Such control is achieved in the Kermeta solution by explicitly processing the classes from leaf classes up to their superclasses. For the declarative languages, a new transformation language construct to specify particular orders of matchings in a model could be useful. Such a facility will be added in the next version of UML-RSDS.

Update-in-place execution: This is directly supported by GrGen, Kermeta and UML-RSDS. In QVT-R this form of execution leads to unclear specifications when deletion is combined with creation and modification, and instead a strategy based on copying the input model to the output model is used. In-place transformations are supported in ATL with the use of the refining mode; however, this execution mode is still immature and only *matched* rules are allowed. A solution with the default execution mode requires model copying as with the QVT-R solution.

Optimisation facilities: In UML-RSDS execution of the constraints is optimised by factoring out repeated occurrences of expressions within constraints using let variables (v in C2 and C3), and by omitting negative application condition tests.

In GrGen, helper operations are used to factor out repeated expression evaluations. In-lining of operation calls can be used to optimise execution.

In Kermeta, repeated evaluation of expressions can be avoided by calling an operation to evaluate the expression once, and storing its result to use in multiple places.

In ATL, *using* definitions (local variables of rules) as well as *helpers* can be used to avoid duplicated expressions.

In QVT-R, subrelations can be defined to factor out some repeated calculations, and the precomputation of sets of elements used repeatedly can be used to optimise computations.

Fixpoint iteration: In UML-RSDS, this is the default implementation of constraints that both read and write some entity type or feature.

In GrGen and Kermeta, it needs to be explicitly programmed as an iteration which halts when no further elements matching the application conditions of any rule exist in the model.

In QVT-R, it is the standard implementation of top-level rules, and it is the standard implementation of ATL matched rules.

QVT-R does not seem suitable for this type of transformation problem, because of the difficulty of handling sets of elements, and the lack of support for input-destructive update-in-place transformations. ATL is not currently suitable because of the limitations of the refining mode. In terms of size, complexity and development time, UML-RSDS and GrGen have clear advantages over Kermeta, and in terms of maturity, GrGen and Kermeta are preferable to UML-RSDS. If guaranteed optimal solutions were required, then Kermeta would be the most appropriate approach to use.

11. Conclusions

The paper has established an evaluation framework and procedure by which organisations can compare and select transformation approaches for a particular problem, using quantitative measures. This can be summarised as:

- Identify a subset or simplified version of the problem, which retains the essential characteristics of the problem;
- Define the external characteristics of concern to the organisation;
- Define the subcharacteristics and internal attributes for these external characteristics, using a standard such as ISO/IEC 9126-1;
- Provide quantitative metrics for attributes, using an approach such as GQM;
- Apply the candidate transformation approaches to the sample problem, and evaluate the measures for the solutions achieved in each approach;
- Select the approach/approaches to be adopted for the full-scale problem based on the resulting rankings of the characteristics of the approaches.

In this paper, the sample problem was simplified by using a small metamodel, compared to UML class diagrams. It nonetheless presents the same key issues as the full restructuring problem.

The comparison on this case study has clearly identified the advantages and disadvantages of different transformation approaches for restructuring/refactoring transformation problems. Specific deficiencies of particular approaches have been identified, such as the lack of appropriate language support in QVT-R and ATL. General deficiencies of transformation approaches have also been identified, such as the need for fine-grained control over the order in which elements of a model are selected for matching against a rule, and intrinsic language support for matching sets of elements at a time.

The results of the paper are specific to the category of update-in-place refactoring transformation problems. Different results and rankings of approaches can be obtained for other categories of problems. In [23] we applied the framework to refinement and migration transformation problems, and observed that Kermeta seems to be the most suitable approach

for small refinement problems, with ATL becoming more appropriate for medium-sized problems. For migration, specialised languages such as Epsilon Flock achieve the highest scores.

Future work will apply the evaluation framework defined in this paper more widely to other categories of transformation problems at different levels of scale. The impact of language style (such as graphical versus text-based specification) upon quality characteristics could also be evaluated. The comparison of approaches on this case study is being continued with further contributions as a transformation case in the TTC 2013 conference.

Acknowledgements

Kevin Lano and Shekoufeh Kolahdouz-Rahimi would like to thank Iman Poernomo and the HortModa EPSRC project.

References

[1] M. van Amstel, C. Lange, M. van den Brand, Using metrics for assessing the quality of ASF+SDF model transformations, in: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations, ICMT '09, 2009, pp. 239–248.
[2] M. van Amstel, C. Lange, M. van den Brand, Metrics for analyzing the quality of model transformations, in: 12th ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering, 2008.
[3] M. van Amstel, M. van den Brand, Model transformation analysis: Staying ahead of the maintenance nightmare, in: ICMT, 2011, pp. 108–122.
[4] M. van Amstel, S. Bosems, I. Kurtev, L. Pires, Performance in model transformations: Experiments with ATL and QVT, in: ICMT, 2011, pp. 198–212.
[5] ATL model zoo, http://www.eclipse.org/m2m/atl/atlTransformations, 2012.
[6] V. Basili, G. Caldiera, D. Rombach, The goal question metric approach, in: J. Marciniak (Ed.), Encyclopedia of Software Engineering, Wiley, 1994.
[7] P. Botella, X. Burgués, J. Carvallo, X. Franch, G. Grau, J. Marco, C. Quer, ISO/IEC 9126 in practice: What do we need to know?, in: Software Measurement European Forum (SMEF 2004), 2004.
[8] F. Buttner, J. Cabot, M. Gogolla, On validation of ATL transformation rules by transformation models, in: MoDeVVa, 2011.
[9] J. Chhabra, Code cognitive complexity: A new measure, in: Proceedings WCE 2011, London, UK, 2011.
[10] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM Syst. J. 45 (3) (2006) 621–646.
[11] Z. Drey, C. Faucher, F. Fleurey, V. Mahe, D. Vojtisek, Kermeta Language Reference Manual, https://www.kermeta.org/docs/KerMeta-Manual.pdf, April 2009.
[12] T. Goldschmidt, G. Wachsmuth, Refinement transformation support for QVT relational transformations, FZI, Karlsruhe, Germany, 2011.
[13] J. Gray, Y. Lin, J. Zhang, Automating change evolution in model-driven engineering, IEEE Computer 39 (6) (2006) 51–58.
[14] R. Gronback, Eclipse Modelling Project: A Domain-Specific Language (DSL) Toolkit, Addison–Wesley, ISBN 978-0-321-53407-1, 2009.
[15] R. Grønmo, B. Møller-Pedersen, G. Olsen, Comparison of three model transformation languages, in: ECMDA-FA, 2009, pp. 2–17.
[16] E. Guerra, J. de Lara, D.S. Kolovos, R.F. Paige, O. Marchi dos Santos, transML: A family of languages to model model transformations, in: MODELS 2010, in: Lect. Notes Comput. Sci., vol. 6394, Springer-Verlag, 2010, pp. 106–120.
[17] ISO/IEC 9126, en.wikipedia.org/wiki/ISO/IEC_9126, 2001.
[18] ISO/IEC, ISO/IEC 9126-1, Software engineering – Product quality – Part 1: Quality Model, 2001.
[19] ISO/IEC JTC1/SC7, ISO/IEC 25010, Software Product Quality Requirements and Evaluation (SQuaRE), 2007.
[20] E. Jakumeit, S. Buchwald, M. Kroll, GrGen.NET: The expressive, convenient and fast graph rewrite system, Int. J. Softw. Tools Technol. Transf. 12 (2010) 263–271.
[21] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A model transformation tool, Sci. Comput. Program. 72 (1–2) (2008) 31–39.
[22] L. Kapová, T. Goldschmidt, S. Becker, J. Henss, Evaluating maintainability with code metrics for model-to-model transformations, in: QoSA, 2010, pp. 151–166.
[23] S. Kolahdouz Rahimi, Measurement and comparison of model transformation approaches using a systematic procedural framework, PhD thesis, King's College, London, 2013.
[24] S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya, P. Van Gorp, Quality improvement case study comparison, http://share20.eu/?page=ConfigureNewSession&vdi=XP-TUe_SCP_ESEiC11_QualityImprovement_Update_Resubmission.vdi, 2012.
[25] K. Lano, S. Kolahdouz-Rahimi, Model-driven development of model transformations, in: International Conference on Model Transformations, 2011.
[26] K. Lano, S. Kolahdouz-Rahimi, Model-transformation design patterns, in: ICSEA, 2011.
[27] K. Lano, S. Kolahdouz-Rahimi, The UML-RSDS toolset, http://www.dcs.kcl.ac.uk/staff/kcl/uml2web, 2012.
[28] K. Lano, S. Kolahdouz-Rahimi, I. Poernomo, Comparative evaluation of model transformation specification approaches, Int. J. Softw. Inform. 6 (2) (2012) 233–269.
[29] K. Lano, S. Kolahdouz-Rahimi, T. Clark, Comparing verification techniques for model transformations, in: MoDeVVa, MODELS 2012.
[30] Medini QVT, http://projects.ikv.de/qvt, 2012.
[31] T. Mens, K. Czarnecki, P. Van Gorp, A taxonomy of model transformation, in: Proc. Dagstuhl Seminar on "Language Engineering for Model-Driven Software Development", Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, 2005.
[32] T. Mens, P. Van Gorp, D. Varro, G. Karsai, Applying a model transformation taxonomy to graph transformation technology, in: Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), in: Electron. Notes Theor. Comput. Sci., vol. 152, 2006, pp. 143–159.
[33] N. Moha, S. Sen, C. Faucher, O. Barais, J.-M. Jezequel, Evaluation of Kermeta for solving graph-based problems, Int. J. Softw. Tools Technol. Transf. (2010).
[34] P. Mohagheghi, V. Dehlen, Developing a quality framework for model-driven engineering, in: MoDELS Workshops, 2007, pp. 275–286.
[35] P.-A. Muller, F. Fleurey, J.-M. Jezequel, Weaving executability into object-oriented meta-languages, in: MODELS/UML 2005, in: Lect. Notes Comput. Sci., vol. 3713, 2005, pp. 264–278.
[36] Object Management Group, Query/View/Transformation Specification, 2010.
[37] Object Management Group, UML Superstructure, version 2.3, OMG document formal/2010-05-05, 2009.
[38] L.M. Rose, M. Herrmannsdoerfer, J.R. Williams, D.S. Kolovos, K. Garcés, R.F. Paige, F.A.C. Polack, A comparison of model migration tools, in: MoDELS (1), 2010, pp. 61–75.
[39] L.M. Rose, M. Herrmannsdoerfer, S. Mazanek, P. Van Gorp, S. Buchwald, T. Horn, E. Kalnina, A. Koch, K. Lano, B. Schaatz, M. Wimmer, Graph and model transformation tools for model migration, in: Software and Systems Modeling (SoSym), 2012.
[40] L. Rose, D. Kolovos, R. Paige, F. Polack, Model migration with epsilon flock, in: International Conference on Model Transformations, Springer-Verlag, 2010.
[41] S. Sendall, W. Kozaczynski, Model transformation: The heart and soul of model driven software development, http://cui.unige.ch/sendall/files/sendall-tech-report-EPFL-model-trans.pdf, pp. 42–45.
[42] System and Software Engineering Lab, Vrije Universiteit Brussel, Belgium, MDE Case Studies, http://ssel.vub.ac.be/ssel/research:mdd:casestudies.

[43] G. Taentzer, K. Ehrig, E. Guerra, J. De Lara, T. Levendovszky, U. Prange, D. Varro, Model transformations by graph transformations: A comparative study, in: Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica, 2005.
[44] P. Van Gorp, P. Grefen, Supporting the internet-based evaluation of research software with cloud infrastructure, Softw. Syst. Modell. 11 (1) (2012), http://dblp.uni-trier.de/rec/bibtex/journals/sosym/GorpG12.
[45] D. Varró, M. Asztalos, D. Bisztray, A. Boronat, R. Geiss, J. Greenyer, P. Van Gorp, O. Kniemeyer, A. Narayanan, E. Rencis, E. Weinell, Transformation of UML models to CSP: A case study for graph transformation tools, in: Applications of Graph Transformations with Industrial Relevance, 2008, pp. 540–565.
[46] A. Vignaga, Metrics for measuring ATL model transformations, MaTE, Department of Computer Science, Universidad de Chile, 2008.