Int J Software Informatics, Volume 10, Issue 4 (2016), pp. 000–000
International Journal of Software and Informatics, ISSN 1673-7288

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661048

DOI: 10.21655/ijsi.1673-7288.00238

Applying Programming Language Evaluation Criteria for Model Transformation Languages

Leila Samimi-Dehkordi, Alireza Khalilian, and Bahman Zamani

(MDSE Research Group, Department of Software Engineering, University of Isfahan, Isfahan, Iran)

Abstract The appraisal of the status quo for the methods of evaluating model transformation languages (MTLs) manifests several shortcomings: they are often either language-specific or feature-specific, they may lack of sufficient discussion on possible values for proposed criteria, few MTLs may be applied in their evaluation, or a combination of these limitations. We have previously proposed a method which utilizes programming language (PL) criteria for evaluation of MTLs. In this paper, an improved method is proposed in which a large family of 11 major criteria with total of 46 sub-criteria, mainly inspired from PL evaluation criteria, is used to evaluate MTLs. Then, an interactive methodology is proposed that consolidates the criteria to establish a *decision-support* system for MTL selection. In order to investigate the effectiveness of the criteria and the proposed methodology, six MTLs were used for studies: ATL, Kermeta, ETL, QVT-O, QVT-R, and TGG. The results of MTL evaluations corroborate that the criteria are highly effective in practice; they provide helpful insights for different users to enable them to choose the most appropriate MTL for the application at hand. With our decision-support methodology, we could have achieved evidence to imply applicability in real-world scenarios.

Key words: model-driven engineering; model transformation language; evaluation criteria

Samimi-Dehkordi L, Khalilian A, Zamani B. Applying programming language evaluation criteria for model transformation languages. *Int J Software Informatics*, **Vol.10, No.4 (2016): 000–000. http://www.ijsi.org/1673-7288/10/238.htm**

1 Introduction

Model-driven engineering (MDE) is an emerging and promising paradigm to the software development in which models are first-class entities and play the key role for the development of software[1,71]. It can be considered as a methodology that helps gain as much benefit as possible from the models in software development.

Similar to the "everything is an object" principle in object-oriented programming that results in simplicity, generality, and uniformity of object-oriented languages, the principle of "everything is a model" in MDE leads to uniformity and coherency of model-driven approaches. For example, the definition of the modeling language itself can be expressed by a model. MDE calls this process *meta-modeling*[72].

Models are not only static or isolated artifacts[1]; but also they can be refined, refactored, merged, aligned, migrated, and improved or even they can be exchanged

Corresponding author: Bahman Zamani, Email: zamani@eng.ui.ac.ir
Received 2016-10-17; Revised 2017-01-03; Accepted 2017-01-23.

between different tools[2]. These tasks can be simply accomplished given the prominent role of model transformations in MDE. Model transformations provide flexible ways to define mappings between models in different types and abstractions. These operations are implemented as model-to-model (M2M) transformations.

In order to express models and transformations, some notations are required which MDE calls them *modeling language* and *model transformation language* (MTL) respectively[1]. Often, for each MTL, at least one tool has been produced and presented. Currently, a large number of different MTLs with various characteristics have been introduced in the literature to enable developers and designers to define transformation rules in a manner that fulfils their requirements as much as possible. The diversity between MTLs and their characteristics prevents them to be employed in general-purpose applications. Hence, we need to choose among different MTLs for each specific task. As a result, the need to the methods for assessment, evaluation, and comparison of MTLs seems necessary.

A large body of literature has established methods to investigate MTLs and the characteristics of MTLs have been studied from different aspects[3,4,5,6,7]. However, there are limitations concerning the current studies:

- checking just a specific characteristic or aspect in different MTLs[5,8]
- specificity to certain language[6] and lack of generalization to all MTLs
- presenting characteristics without any discussion on possible values for the defined characteristics[2]
- evaluating a few (possibly non-representative) MTLs
- receiving less attention by Triple graph grammar (TGG) as compared to others
- applicability for evaluation of some candidate MTLs for a certain task; After evaluation completes, developer is responsible for final selection of an MTL according to the results of evaluation. This task might be challenging and draws an important gap in current MTL research.

These limitations motivated us to identify a family of criteria which we believe have much potential to evaluate MTLs more precisely. We have previously proposed a method in which programming language (PL) evaluation criteria were used for the assessment of MTLs[9]. This method used three major criteria and nine sub-criteria for the task of MTL evaluation and studied the effectiveness of the method using five MTLs. The results were encouraging and served us as a proof of concept.

In this paper, we largely extend the mentioned research work by proposing a family of 11 major criteria and 46 sub-criteria mainly used in the assessment of PLs. These criteria were adapted to utilize their potentials for the task of MTL evaluation which has very similar properties and requirements. This idea roots in the intuition that MTLs are also languages to express computations. The only (slight) difference lies in their abstraction level and the type of computations they do. Specifically, we intended to compile the criteria with the following properties:

- the criteria that focus on evaluation of MTLs not their tools

– the criteria that the evidence to measure them can be easily found; everyone can make a justifiable measurement on the criteria and the results would be reproducible.

Given that PLs have much longer precedency than MTLs and their characteristics have been investigated in high number of studies[43] using the various specific designed criteria, we can adapt them for evaluating MTLs. On the one hand, long background behind PL evaluation criteria along with deeper and more comprehensive understanding of their interrelationships and mutual effects facilitate the measurement task. On the other hand, specificity and suitability for the assessment of PLs themselves, makes them an appropriate choice to MTL evaluation and satisfaction of our design goals. These features also make measurements of the criteria simple yet more accurate despite the fact that most of them are subjective. The PL criteria we introduce are not novel themselves; however, applying them in MTL evaluation is a novel idea.

In addition, we employed the criteria to establish a systematic methodology to recommend an appropriate MTL in a given application. The proposed methodology takes the stakeholders of an application along with their requirements and priorities (preferences) as input. Then, it uses the criteria to compute the score of each candidate MTL and suggests one with the highest score. The novelty of this methodology is to establish a *decision-support system* considering *stakeholders* of an application, their *requirements* which are a reflection of the underlying *task*, and their *preferences* which is a reflection of their *experience* in applying MTLs. Combining these pieces of knowledge, the methodology ranks candidate MTLs and suggests the most conforming one. The proposed methodology is *general* and *flexible* enough to be used with any evaluation method using different criteria.

In order to investigate the effectiveness of the criteria, we have conducted a comprehensive study on six transformation languages as per following: AtlanMod Transformation Language (ATL)[10], Epsilon Transformation Language (ETL)[11], Kermeta[12], QVT-Operational (QVT-O)[13], QVT-Relational (QVT-R)[13], and Triple Graph Grammar (TGG)[14]. As expected, the results confirm that no MTL is absolutely superior to the others. More importantly, the results show that there is potential benefit to be achieved by accounting for the PL evaluation criteria to the assessment of MTLs. In other words, these criteria provide deep understanding of the underlying characteristics of MTLs from various aspects. This recognition helps the user pick the most appropriate MTL among some candidates for his/her specific application. In summary, the major contributions of this paper are as follows:

– A family of criteria inspired from PL evaluation criteria which are accorded for the assessment of MTLs

– A systematic requirement- and priority-aware methodology to recommend the best-fit MTL for a certain task, taking into account the stakeholders of a task, their requirements and their preferences

– A comprehensive study evaluating six MTLs using the criteria along with a detailed discussion on the results

The rest of the paper is organized as follows. Section 2 presents discussion on language selection for our evaluation. Section 3 presents the criteria for the assessment of MTLs. In Section 4, the results of evaluating six transformation languages using the introduced criteria are shown in detail. Section 5 gives a discussion on the criteria and the results. Related work is expressed in Section 6. Finally, Section 7 concludes the paper.

2 Language Selection

In order to establish an evaluation on presented criteria and applying our proposed methodology, we need to choose some MTLs. The chosen languages should be as much *representative* of large number of current MTLs as possible to help the results generalize. M2M language selections are often made for pragmatic reasons, e.g., availability of a local expert, a similar problem solved using that language, and so on. One way to determine suitable MTLs for our study is to review the major body of literature. By investigating previous similar studies, we found six languages that are common, widely-used, and well-known both in academic research and industrial practice. These six MTLs are listed in the following along which we give the specific reason behind selection of each MTL.

- ATL[10,19] is a widely-used and well-known MTL which is used as a *benchmark* in most studies.
- ETL[11,20] is a *hybrid* rule-based MTL that supports *declarative* rules with *imperative* bodies, logic OCL expressions, Java object method calls, and so on. It also presents all of the standard features of an MTL along with the capability of transforming many input to many output models[18].
- Kermeta[1][12,21] is a *general-purpose* MTL similar to PLs.
- Two QVT languages: QVT-O[22,23] and QVT-R[13,24,25,26] are *standardized* MTLs established by Object Management Group[2] (OMG) and have been commonly evaluated in different studies.
- TGG[14,27,28,29,30,31] is a popular and widely-used *representative of graph grammar* MTLs with less attention in studies. To note that TGGs are a category of approaches themeselves and may seem to be out of place compared with other languages that are concrete instances of languages. Therefore, here TGG is an abstract MTL consisting of common characteristics of its concrete instances.

In order to evaluate a certain MTL, one needs to know the *syntax*, *semantics*, and *pragmatics* of that language. However, to save space and because we investigate the characteristics of each selected MTL when evaluating using our criteria, we postpone further explanations about languages to Section 4 where is best positioned for this task.

[1] `www.kermeta.org/`
[2] `www.omg.org`

3 The Proposed Method

While PLs and MTLs share many commonalities, they differ at least in two aspects: the abstraction level and the primitives or alphabets used for programming. Both PLs and MTLs are used for some kind of computation by computer. The *program* written by each category is realization of an algorithm which is typically interpreted or compiled and executed. However, the level of abstraction in MTLs is often higher than that of PLs. The concepts such as model, meta-model (MM), transformation rules, and conformance are typical elements of programs written by an MTL. By contrast, variable, memory cell, definition and declaration, parameter passing, registers and segment-offset address (in lower level languages), and many others are primary concerns in PLs. Model in an MTL is more abstract than a variable in PL. Some elements including loops, conditionals, and selections are common between PLs and MTLs. The mentioned elements of PLs and MTLs in turn comprise their primitives or alphabets which are used by programmer to construct the program. Moreover, MTLs are often application-specific as compared to PLs that are typically general-purpose. MTLs are designed and customized (and possibly optimized) for the task of model transformation. In contrast, PLs are designed to establish arbitrary computation.

Suppose that C_P and C_M are computations for which PLs and MTLs are *conventionally employed* respectively (not computations that can theoretically be carried out by PLs and MTLs). Then we can see that for C_P and C_M, neither is superset or subset of the other. But rather we have $C_P \cap C_M \neq \varnothing$. Similarly, suppose that A_P and A_M are alphabets, primitives or basic elements and constructs, of PLs and MTLs accordingly. Again we observe that for A_P and A_M, neither is superset or subset of the other and we have $A_P \cap A_M \neq \varnothing$. The extent of intersection for each couple is very high though they hardly equate.

This amount of commonalities between PLs and MTLs motivates us to exploit PL evaluation criteria for the task of MTL evaluation. The only requirement is adaptation or re-definition of PL criteria for MTL context. In the process of re-definition, we should take the (higher) level of abstraction of MTLs and their constituting primitives into consideration.

Another problem that merits explanation is subjectivity of evaluations. Most of the evaluation criteria are wide concepts and their values are ambiguous[32]. It is difficult to find two computer scientists that agree on the meaning of evaluation criteria and their values[32]. Therefore, it is reasonable to assume that most evaluations on software are by nature subjective. Objective measurement needs provable facts which are not the case for evaluation criteria with vague meaning and values. Nevertheless, the research community attempted to mitigate this issue by incorporating objective measurements[2,3,4,35]. This will help limit the threats concerning subjective evaluations. There might be rare situations where evaluations are too subjective to final decision on MTLs. Even in these cases, evaluation criteria provide useful insights in recognition of MTLs. This is because they are designed to help the developer to look at MTLs from different perspectives. Hence, they would facilitate decision making on MTLs for the developer.

Measuring of subjective criteria can be performed in several ways. For one, we can define a set of objective sub-criteria and measure them. This is the case in this

paper. For two, we may provide questionnaires and ask experienced persons. This might be impractical because access to such persons is not always trivial. For three, a comparison method for each individual criterion is employed; i.e., MTL x is better than y. However, the amount of superiority of x cannot be measured. Finally, we can provide some categories and their features. Then, determine a score for each category and MTLs are scored according to the category they belong. This makes comparison relatively hard although we favour it in some situations.

We emphasize that the presented criteria are intended to evaluate MTLs themselves not programs. In other words, evaluating MTLs is assumed to be independent of the problem that is being solved. A source of evidence to this claim is that even with a well-designed language, a programmer may produce a program that is unstructured and less readable and reliable[32]. The size, complexity, and nature of the underlying task does not affect to the particular features of an MTL itself such as readability. It does influence on the solution produced by developer due to, for example, bad design, lack of enough experience, misuse of language constructs, and so on.

3.1 The evaluation criteria

Here, we present the criteria: *readability*, *writability*, *reliability*, *maintainability*, *learnability*, *generality*, *portability*, *reusability*, *availability of tools*, *standardization*, and *cost*. Figure 1 depicts the criteria and their interrelationships.

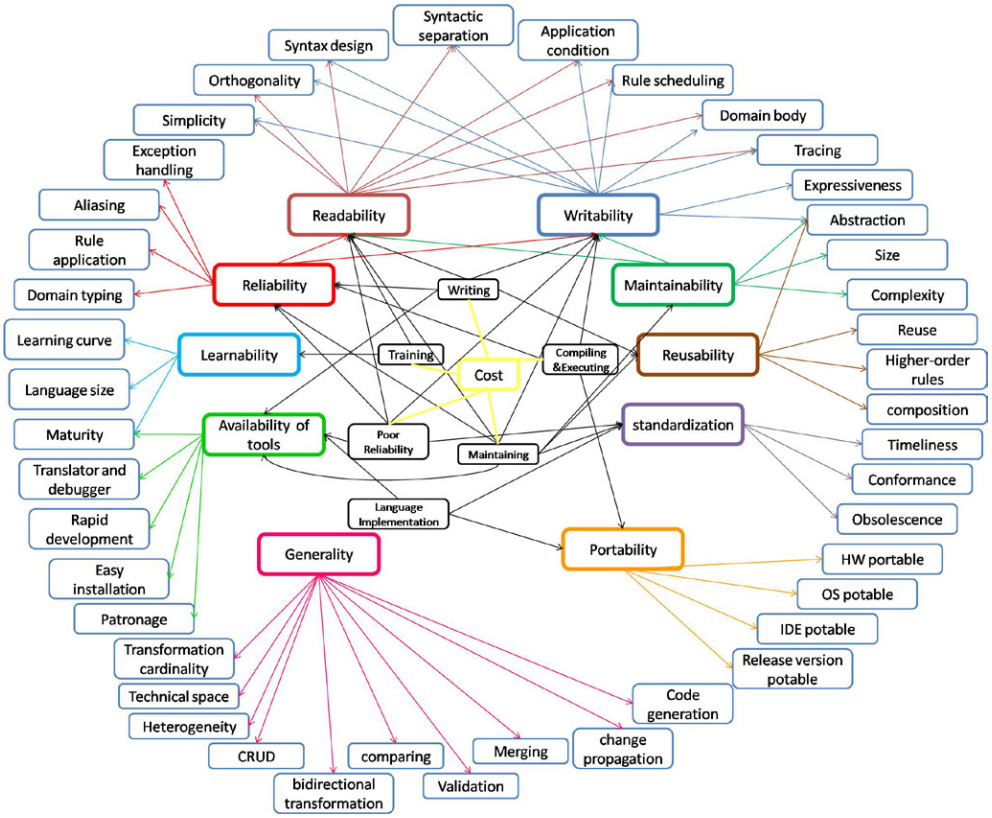

Figure 1. The criteria and their relationships.

3.1.1 Readability

Readability is the ability to understand what has been written, which is defined based on some PL characteristics[32] and transformation language features[33]. Evaluation of readability of a language must be accomplished in the context of its application. More precisely, comparing the readability of two languages with different application domains is not justified. The characteristics and features of readability are described in detail in the following.

Simplicity: A language with a small set of basic constructs and primitive features is a simple language[32]. It is desirable to have minimal *operator overloading* in the simple languages[32]. From the model-driven paradigm point of view, declarative languages are simpler than imperative ones. The hybrid MTLs combine the simplicity of declarative approaches with the expressive power of the imperative ones to utilize the benefits of both styles. We define *simplicity* based on the *number of primitive types* and minimal *operator overloading*.

Orthogonality: A language is orthogonal when a small set of basic constructs can be combined in a relatively small number of ways to create data structures of the language[32]; every possible combination of relatively small number of primitives should be allowed in the orthogonal language and have a meaningful construct. In the context of MTLs, supporting *parameterization*[33] can be considered as one of the orthogonality characteristics, which has three features:

(1) *Control parameters*, (2) *Generics*, and (3) *Higher-order rules*. Here, orthogonality deals with combination of rules and the parameters that are allowed to be sent to them and whether rules are allowed as input parameter to the other rules. For the case of PLs, this situation occurs in functions instead of rules. In fact, the difference lies in the notion of abstraction level which is argued at the beginning of Section 3.

Control parameters allow implementing policies by passing values as control flags. *Generics* allow passing data types as parameters, which includes model element types. *Higher-order rules* are the rules, which take other rules as parameters. Another significant feature of orthogonality is *data type completeness*, a principle[34] that states all of the data types should have the same status; "no operation should be arbitrarily restricted in the types of its operands[34]." If a language has some degrees of class distinctions in its data types, it violates type completeness.

Syntax design: The language syntax has a significant role to make a language readable. The examples of syntactic design choices are *keywords*, *method of forming compound statements*, and *form and meaning*[32]. One of the critical issues of keywords is whether they can be used as variable names. If it is possible, the transformation code will be confusing, leading to low readability. Moreover, increasing the number of keywords improves the readability of programs. Form and meaning is described by statements that their appearance indicates their semantic purposes; the syntax (form) should indicate the semantic. The language should prevent identical constructs with different meanings. More precisely, the language should be grammatically transparent; a one-to-one mapping between grammatical form and semantic meaning. Note that syntax and structural design of a language does not necessarily imply high readability of programs. Readability should also be considered in the context of a language's application.

Syntactic separation: If the operating parts of the source model are separated from the operating parts of the target model in a transformation rule, the language supports syntactic separation[33]. It is obvious that supporting syntactic separation makes the transformation more readable. The degree of the separation is described based on the language styles; for example, in TGG, the operating parts of source, target, and corresponding graphs are completely separated. However, in imperative languages such as Kermeta, this separation is not considered.

Application condition: It is a condition on a rule which controls its execution[33]. In order to execute the rule, its *application condition* must be true. This feature enables the user to define some rule for special cases of the model elements. It can be described as a kind of control construct, which is the main characteristic to make a language more readable[32]. Lack of this feature leads to, for example, multiple if-else that results in less readability.

Rule scheduling: Scheduling is a mechanism, which determines the order of applying individual rules. In some languages, it is possible to schedule transformation rules explicitly[33]. In this case, the programmer can read the transformation rules in a meaningful order. Explicit scheduling can be defined internally or externally. The external scheduling separates the rules from the scheduling logic. However, in the internal manner, the scheduling is described directly in the rules by invoking other rules. The implicit scheduling makes the language less readable.

Domain body: A domain is a rule part, which is responsible for accessing one of the models[33]. In each application more than two source and target domains are allowed. Domain body has three subcategories including *variables*, *patterns*, and *logic*[33]. In the *variables*, the source or target elements are kept. *Patterns* are the fragments of models containing some variables. In some cases, they also contain meta-language expressions and statements. Based on the model representation, patterns can be categorized in *string*, *term*, or *graph* patterns. Model to model transformation approaches usually apply term or graph patterns. Graph patterns make language more readable, and string patterns make it less concise. Patterns are described by the *abstract* or *concrete* syntax of the source or target MMs. The concrete syntax can be *textual* or *graphical*. Graphical syntax is clearer than textual in reading. *Logics* are defined as the computations or constraints on the model elements which are determined based on the way of *value specification* and *element creation*. There are three alternatives for the value specification: *imperative assignment*, *binding values*, and specification values through *constraints*. Elements can be created *implicitly* or explicitly. Explicit creation makes the program more readable.

Tracing: Tracing is defined as runtime footprints of execution[33]. In the trace model, there are trace links which connect source elements to the target ones. Trace data can be helpful to know how modifying one model affects other related models which are useful for impact analysis, model synchronization, and transformation debugging. The traceability links can be recorded in a separate model, or in one of the source or target models. If traceability links are stored in a separate model, the program reader can follow the results of the transformation easier.

3.1.2 Writability

Writability is the ability to say what you mean without any expressive verbiage.

Writability is provided by means of regular, concise, short, and cryptic structures. Most of the features affecting readability would also influence writability because when writing a transformation code, it is read many times to write the remainder or modify the existing parts. Note that writability must also be evaluated in the application domain of a transformation language. In the following, the sub-criteria of writability are explained.

Simplicity and orthogonality: Some features of a language with a great number of constructs may be misused or disused because of the programmer unfamiliarity and misunderstanding. Therefore, a relatively small set of basic constructs and a consistent number of combination ways is much more preferable to support writability.

Syntax design: If the language syntax is compatible with the language semantic, writing a transformation program can be easier. In this case, the programmer does not need to remember identical constructs with different meanings. Existence of default or implicit grammatical rules in a language helps increase the writability of programs.

Abstraction: The ability of defining complicated structures (or operators) at higher levels such that unnecessary details can be ignored is called abstraction. If a language supports a high level of data and control abstraction, it will be more writable; in this case, more complicated computations can expressed shorter and simpler[32]. In the model-driven context, abstraction is determined based on supporting specification of *abstract rules*, *operation overloading*, *simplification*, *selection*, *generalization*, *reflection*, *aspect*, and modularity. Abstract rules, which are not executable, can be used to specify core behavior for the sub-rules[35]. The capability of specifying user-defined operations (function), and operator overloading can lead to abstraction[35]. Simplification is denoted as abstracting the language-specific details and abstracting from control flow[36]. Abstraction by selection concentrates on a particular part of the MMs to enable the developer to define transformations in a divide-and-conquer manner[36]. Generalization process is defined based on the generic transformation logic, which is supported by generic types[36]. In reflection, transformation rules can access the transformation themselves[33]. Aspect enables the language to express concerns crosscutting several rules[33]. Modularity is a mechanism of packaging rules into modules, which can be imported in other modules[33]. Each module consists of a set of objects including procedures, data types, variables, and other artifacts that constitute an abstract unit. Modularity is essentially dependent to information hiding[37] which refers to hiding objects and computations from other portions of the system that do not require them. Note that modularity also contributes much to the handling of complexity in developing large applications.

Expressiveness: Expressiveness, expressivity, or expressive power has a broad range of meaning. In the context of PLs, expressiveness means to provide convenient, natural, and appropriate ways to express computations and algorithms. A language would be more expressive if it has syntactical structures that programs written in that language reflect the logical structure of the algorithm clearly. In the context of formal languages, expressiveness refers to the language's power in expressing ideas and computations; the notion of *Turing-complete*. For the case of transformation

languages, it is common to consider expressiveness as Turing-completeness[38].

However, we can benefit from the both worlds: convenient ways to express computations and Turing-completeness. A Turing-complete language should support *unbounded memory*, and *unbounded repetitions*[38]. If a language only allows bounded iteration or recursion, it is non-Turing-complete because the termination of the transformation execution becomes decidable. It is worth mentioning that for visual languages, the looping constructs are defined as blocks with internal repetition and blocks applied on collections. Recursion in visual languages is presented by control flows, which are transferred back to a previous block.

Bidirectionality is a valuable feature for MTLs which has significant impact on expressiveness power of an MTL. It refers to the capability of a language in which a rule is interpreted from both source to target and from target to source provided that a single source and a single target model are used[16].

Finally, *incrementality* is another influencing factor in expressive power of a language. It is the ability of updating one existing model based on the changes in another[33]; that is, the former model would not be computed from the scratch. This feature can enhance the performance when the source and target models are very large. Expressiveness is influenced by incrementality in that it helps express some applications in a more natural and convenient manner.

Syntactic separation: As mentioned, syntactic separation is one of the features of the readability and it also affects the writability of the transformation program. For instance, in the graph transformation, the operating part of the source is separated from the target part, and this separation can lead to high writability in the program or model of the transformation. In other words, it makes the transformation rules simpler and more writable.

Application condition: The conditions for restricting the model elements, which are operated by the rule, make the language more expressive due to providing natural ways to express transformations. Using application conditions often leads to eliminate *if* statements. Therefore, a program can be written more concisely increasing writability.

Rule scheduling: If a language provides scheduling implicitly, transformation programs by this language can be written simpler. However, for readability it is preferable to define the rule scheduling explicitly. This is one of the features that make readability and writability conflicting.

Domain body: If a language has a graphical syntax, writing transformation programs can be simpler. For element creation if it is performed implicitly, the writability can increase. Imperative assignment is more natural for specifying value in comparison to value binding.

Tracing: If the creation of traceability links needs to be manually encoded, the writability of the transformation language decreases. For automatic tracing, some control may also be required to determine what should be recorded.

3.1.3 Reliability

Reliability is the ability of checking erroneous assumptions in the source model and handling the exceptions[5]. A model transformation program is reliable if it always works according to its intended specifications. It is determined based on PL criteria

including readability, writability, exception handling, and aliasing, and transformation language features, i.e., rule application strategy, and domain typing.

Readability and writability: Reliability is much influenced by both writability and readability. Programs written with unusual and unnatural structures are less likely to be reliable in all situations. In other words, if a language lacks the natural ways to express computations, programs have to use unusual structures and ways of programming which results in a program that may not work properly in all situations. Readability also influences the reliability of the transformation in implementation and maintenance phases. Programs that are hard to read or have low readability due to some features are difficult to analyze and to modify.

Exception handling: In some cases, during the transformation execution, exceptional situations may be happened. The ability of intercepting runtime errors by encoding some statements in the program is called *exception handling*[32]. In model transformation programs, an exception can be handled explicitly by user with defining exceptional situations and their handlers in the transformation. In other cases, exception can be handled implicitly in the implementation of the transformation language, or may not be handled at all[5].

Aliasing: Having more than one distinct access method to the same memory cell is called *aliasing*[32]. Pointers to the same variable are an example. Less reliability and difficulty of program verification are the major issues.

Rule application strategy: Each rule should be applied in a certain location in its source scope. In a certain source scope, there may be multiple matches for a rule. This is why we need to a strategy to specify the application locations. This strategy can be deterministic, nondeterministic, or interactive[33]. Deterministic strategy leads to high reliability.

Domain typing (Type checking): The typing of the domain body components can be *untyped*, *syntactically* typed, or *semantically* typed[33]. Template-based approaches usually use untyped patterns, which are not considered in this paper. Most of the M2M transformations are syntactically typed, which means that compatibility of the data types in expressions are checked at compile-time. In the context of PLs, it is called static type checking. In addition, syntactically typed languages can be further divided into *strongly-typed* or *weakly-typed*[37]. If every type incompatibility or type error of expressions is detected at compile-time, then we would have a type-safe language and it is considered as strongly-typed. Otherwise, if there is at least one situation which is not type-checked at compile-time, the language would be weakly-typed. Strong typing is an important property for a language, in that it highly increases the reliability of the programs. In the semantic typing, well-formedness rules or behavioral properties are allowed to be asserted.

3.1.4 Maintainability

This is the ability of implementing a specified modification to the transformation program effectively[32,39]. It is determined by the *size*, *complexity*, *readability* of the transformation and *abstraction mechanisms* employed in it.

Size: The *size* of transformation is determined by the number of code lines on the same problem. The larger the size and the complexity of transformation program,

the lower the transformation *flexibility*.

Complexity: The *complexity* is determined by the total number of occurrences of operators, features, and entities and the number of operation and rule calls on the same problem.

Readability: Since maintenance stage includes modification and/or extension of the existing code, one must be able to read and understand the code easily. High *readability* dramatically reduces the costs of maintenance stage and overall lifecycle of transformation program.

Abstraction: Correct application of any kind of *abstraction* within the transformation program impacts on its readability and facilitates maintenance tasks. Since abstraction helps ignore unnecessary details, the complexity of the overall transformation reduces. Hence, any modifications and extensions on transformation program are made easier. A properly-defined abstraction implies encapsulation of data and operations. Encapsulation helps reduce the complexity and increase understandability of programs. The net effect is facilitating the tasks of maintenance.

3.1.5 Learnability

This criterion includes features that help programmers learn the language more rapidly or factors with strong effects on simplicity and thus learnability of a language[40]. Three sub-criteria can be considered:

Learning curve: The learning curve is the time that takes to be mastered the basics of the language. In other words, it is defined as the expected difficulties in learning to read and write language specifications[41]. Low learning curve leads to high learnability. If a language constructs are defined based on another well-known language, the ease of learning of the former will increase.

Language size: There exist at least two cases for a language in terms of its size[34]. If the language's kernel includes all the required features of a programmer, it is *monolithic*. Otherwise, it is *microkernel*. For the case of microkernel languages, the language itself has a small number of data-types and constructs. However, strong abstraction and rich library of reusable modules are provided for different applications. The programmer can also define any required structure or data-type and underlying operations of his/her own using the primitives of the microkernel language. The result is transferring the complexity from the language itself to its library. Besides, the learnability of the language increases. Monolithic languages have built-in and predefined features, which can hardly be extended. So in a special application, the language is not able to provide the requirements exactly. On the other hand, users of microkernel languages can skip the existing libraries of the language and produce their own libraries according to the specific requirements.

Maturity: Maturity of a tool is defined as the history of use or the number of years that the tool has been public, the number of case studies implemented by the tool[17], and the availability of technical supports such as forum.

3.1.6 Generality

Generality is the ability to use the language in a wide range of applications. Generality has conflict with simplicity because it needs several features to make the

language applicable in many situations. If a language can define transformations to apply the following features[15], we consider it as a general language:

Transformation cardinality: It is related to the number of source models that is allowed in the transformation and the number of target models which are generated as output. There are four cases for the transformation languages including 1-to-1, M-to-1, 1-to-N, and M-to-N, the latter case makes the MTL more general.

Technical space: This refers to the family of technologies, such as data structures, parsers, file formats, and data manipulation facilities to represent models[16]. The examples include XML, XMI, and EMF.

Heterogeneity (Endogenous vs. Exogenous): Endogenous transformations include those with the same source and target MM, while exogenous transformations work on source and target models with different MMs.

Ability to create/read/update/delete transformations (CRUD operations): If a language provides all of the mentioned operations, its expressive power will increase and it becomes more general.

Ability to specify bidirectional transformations: Bidirectional languages use a single source and a single target model and allow interpretation of rules in both ways.

Support for traceability and change propagation: The ability of providing a log of the execution of transformation which can be either built-in into the tool or can be implemented as part of the transformation itself[16]. Besides, the ability to transform only the changed part of the source model in case of any change is called change propagation.

Model merging: It refers to merging the data of several models, with same or different MMs, into a common model.

Model comparing: One of the requirements of the model-driven area is model comparing which is used in some transformation case studies. Support this feature leads to more general languages.

Model validation: One of the applications in the model-driven context is model validation. Some transformation languages can provide techniques to validate generated models, or input ones. In other cases, transformation languages can be integrated to validation languages.

Integration to code generator: This paper studies M2M transformation. However, another necessary type to separate Platform Specific Model (PSM) from Platform Independent Model (PIM) is model-to-text transformation. In most cases, the transformation languages are designed either for M2M or for model-to-text transformation. However, some facilities are required to integrate these two types of transformation languages to support more applications.

3.1.7 Portability

Portability is the ability of moving a program from one implementation to another without any changes in the transformation program[32]. It can be defined in the different levels of platform layers including level of *hardware*, level of *operating system*, level of Integrated Development Environment (*IDE)*, and level of *release version*. For the most transformation language environments, which are implemented as an Eclipse plug-in, only the level of release versions can be considered. It is worth to mention

that Eclipse can run on both Linux and Windows. However, Eclipse has different versions which may not support a special plug-in on some releases.

3.1.8 Reusability

Reusability is the extent to which a transformation program or parts of a transformation program can be reused in other applications[39]. It is defined based on two mechanisms: abstraction and reuse. Composition can be defined as one of its characteristics. Reusability can be affected by supporting higher-order rules[35].

Abstraction: As stated, abstraction is one of writability sub-criteria. It is defined based on eight features, i.e., abstract rules, operator overloading, simplification, selection, generalization, reflection, aspect, and modularity that all of them affects reusability. For example, modularity allows a module to import other modules; it is considered as a kind of reuse.

Reuse: Reuse mechanism is the ability of specifying a rule (or a module) based on using other rules (or modules). Inheritance between rules such as *rule inheritance* is another feature of reuse mechanism. In some languages, the multiple inheritance is supported. *Unit inheritance* increases the reusability of the language.

Composition: This feature, in which a rule is invoked from another rule, is used to compose transformation rules that are a kind of reusing process[33].

Support for higher-order rules: *Higher-order rules* are the rules, which reuse other rules as parameters[35].

3.1.9 Availability of tools

This criterion refers to the existence of good-quality translators and integrated development environments (IDEs)[34]. Rapid development and easy installation make the tools for the language more attractive. One of the significant factors that affect language success is *patronage* which means having a powerful sponsorship for a language[37]. *Maturity* which is discussed in the learnability criterion is another investigated feature.

Translator and debugger: A good-quality translator checks the language's syntax and type system, translates and runs the source program efficiently, and reports any errors precisely. A good-quality IDE combines all aspects of documenting, helping, developing and *debugging* of programs effectively.

Rapid development: It is defined based on the frequency of release and regularity in bug fixing.

Easy installation: In *easy installation*, the language tool can be installed easily without any other requirements. Therefore, it is determined based on the number of plug-in or software that are required to be installed.

Patronage: In fact, the amount of usage and applicability of a language and its lifecycle depends in large to whether it is built, developed, and sponsored by a famous and well-fixed organization or company, even if languages with better design and constructs are available[37]. This patronage to the language and dominance to programmers will result in high inertia; programmer's expertise and producing large amount of software. This situation makes the replacement to the language even more difficult. Therefore, sponsorship and the underlying problems associated with it are more affective to the applicability of a language as compared to technical ones. For

example, QVT languages owe their life to OMG.

3.1.10 Standardization

In order to make the implementation of languages compatible, standard definitions for each language are provided[42]. Standards of a language are either *public*, which are defined by international organizations, or *private*, which are defined by the owner and constructor of the language. Public standards are preferred to private ones. Standardization is related to three factors including timeliness, conformance, and obsolescence[42].

Timeliness: It refers to the appropriate time for standardization of a language. A language may be *never standardized*, *early-standardized*, *lately-standardized*, or *timely-standardized*. A language is early-standardized if it is standardized before releasing any implementations of that language. A language is lately-standardized if is standardized after releasing a number of translators for that language, which leads to incompatibilities. A language is timely-standardized if the standardization occurs after few years of language design and releasing limited number of translators. This makes useful experiences to the language strengths and weaknesses. Consequently, a well-designed language would be finally released.

Conformance: This refers to the fact that current translators conform to the standard of the language or not.

Obsolescence: This criterion shows whether the language's standard is reviewed periodically or not. Due to the ongoing new requirements, the language's standards should be reviewed and upgraded in certain periods. A major problem deals with the existing developed programs. Modification of existing programs to make them compatible with new standards is often time-consuming. Hence, new standards should preserve the functionalities of current standards.

3.1.11 Cost

The cost criterion is by nature different from other criteria and is defined based on several characteristics[32]; moreover, the impact of the mentioned language criteria on different aspects of cost is shown in Table 1.

Table 1 Different aspects of cost and affecting criteria

	Training	**Writing**	**Compiling and Executing**	**Language Impl. System**	**Poor Reliability**	**Maintaining**
Readability	✓	✓			✓	✓
Writability		✓	✓		✓	✓
Reliability		✓	✓		✓	✓
Maintainability						✓
Learnability	✓					
Portability			✓	✓		
Reusability		✓				✓
Availability of Tools		✓		✓	✓	✓
Standardization				✓	✓	✓

Cost of training: This is described based on the readability and learnability of

the language. It also depends on the programmer's experience. The cost of training is strongly dependent upon whether it is based on a well-established popular language or has a new structure. For the first case, many programmers are familiar with the overall structure and properties of the language and training will be completed at shorter time with lower cost.

Cost of writing programs: This cost is mainly dependent on writability of the transformation language and the underlying tools and programming environments. However, readability and reusability are other influencing factors.

Cost of compiling and executing: This type of cost is influenced by syntax design, reliability, and portability. Syntax design affects on the design and cost of translator. Reliability features usually increase execution costs due to runtime checks. Portability often necessitates some kind of interpretation which would further increase execution cost.

Cost of the implementation system: Free translators of a language can be one of the factors to the acceptance and popularity of a language. A language with a heavyweight implementation system, expensive hardware, or special software requirements cannot be used widely. If the language tool is free and open source, and it is portable, we consider it less costly.

Cost of poor reliability: This cost is a function of readability, writability, reliability, availability of tools, and standardization. Standard definitions and specifications result in consistency of the programs in different platforms.

Cost of maintaining programs: This cost is a function of corrections and modifications to add new functionality. It is dependent to several features of a language including readability, writability, reliability, maintainability, reusability, availability of tools, and standardization. Among these features, readability is more important.

3.2 Interactive and quantitative evaluation

Particularly, when a software company or organization needs an MTL for a specific task, some candidate MTLs may be chosen. On the other hand, there exist various stakeholders such as designer or project manager with similar or different preferences or requirements. For example, from the quality assurance analyst point of view, exception handling mechanisms may be of high importance. In contrast, however, the developer or designer might prefer a language that is more writable. As a consequence, diversity among requirements of different stakeholders makes the choice of an appropriate MTL a challenging task. In other words, an overriding question arises that, which MTL should be chosen to satisfy most of the stakeholders and best fits their requirements. This issue motivated us to propose an interactive flexible methodology that can be adjusted to quantitatively evaluate candidate MTLs. This methodology considers several underlying parameters efficacious in the assessment of MTLs and recommends the language with the highest score. Different preferences of stakeholders often reflect their *experience* in applying an MTL. Their requirements also reflect the underlying *task* at hand. Therefore, the novelty of the proposed methodology is to exploit these pieces of information together to assist in MTL selection. Another side of its novelty is that it is general enough to be used with other evaluation criteria which employ different

sets of criteria. It can be used as a *customizable decision-support system* in the context of MTLs. Figure 2 shows the process at high-level.

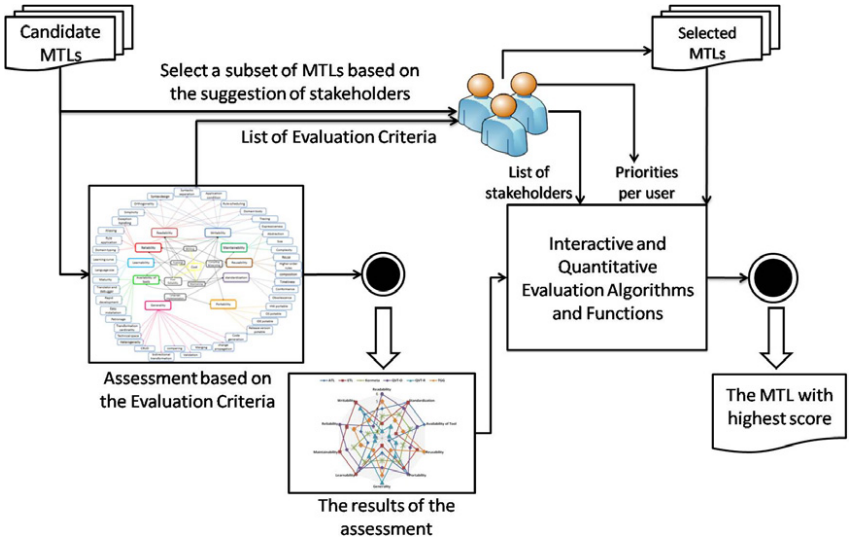

Figure 2. The overall decision-support methodology for MTL selection.

Given an application for a development team with model transformation tasks, the exact steps of the process behind the proposed methodology can be summarized as follows:

STEP 1: Various stakeholders in the team that are involved in working with model transformations are identified. Designers, programmers, project managers, and analysts are some examples.

STEP 2: Some MTLs according to the suggestion of stakeholders are nominated.

STEP 3: Considering the mentioned criteria and their sub-features, the requirements of each stakeholder are determined. In particular, we define five levels of priority (importance) for each criterion or sub-feature: *inoperative*, *weak*, *typical*, *high*, and *strong*. For each stakeholder, the level of priority behind each criterion and each of its sub-features are specified. These priorities then act as *weights* of criteria and can be extended or reduced as needed.

STEP 4: A qualitative assessment of the candidate MTLs are carried out using the mentioned criteria. The steps 5 to 9 are repeated for each stakeholder.

STEP 5: The priorities are mapped to real numbers of the range [0, 2]. Specifically, we simply assign the vector [0, 0.5, 1, 1.5, 2] to the levels of priorities [inoperative, weak, typical, high, strong] accordingly. This assignment is natural and intuitive. By default, the priority of every criterion is set 1.

STEP 6: The results of the qualitative assessment should be mapped to real numbers of the range [0, 1]. To obtain the possible values of each sub-feature, we divide the length of the range by the number of possible cases.

For example, four possible cases yield the values 0, 0.33, 0.66, and 1. In order to map the quantitative values to qualitative cases, lower values are assigned to less desirable cases. This choice is also delegated to the stakeholder to maintain the full coherency of the interactive methodology. For the case of sub-features whose values

are countable or vary in a wide range, such as the number of keywords, we need to discretize them. More precisely, the overall range is partitioned into sub-ranges and each sub-range is classified with a qualitative label.

STEP 7: Using the Eq. (1), the score of each criterion per candidate MTL is computed. Eq. (1) can be treated as a *recursive* function if a criterion consists of multiple sub-criteria; each sub-criterion in turn consists of several sub-sub-criteria, and so on. In this scheme, each of sub-criteria in any level can also have different priorities.

STEP 8: Using Eq. (2), the overall score of each language is computed by combining the scores of each criterion.

STEP 9: At this stage, the language with the highest score is recommended to the related stakeholder. Such language has the highest *conformance* to the preferences and requirements of the corresponding stakeholder.

STEP 10: Finally, the majority voting is used against the languages with highest score which were recommended to each stakeholder, to choose the language that maximizes the number of stakeholders.

$$Score(C_{i,j,k}) = \frac{\sum_{l=1}^{m} w(SF_{i,j,k,l}).v(SF_{i,j,k,l})}{\sum_{l=1}^{m} w(SF_{i,j,k,l})} \tag{1}$$

$$Score(L_{i,j}) = \frac{\sum_{k=1}^{n} w(C_{i,j,k}).Score(C_{i,j,k,})}{\sum_{k=1}^{n} w(C_{i,j,k})} \tag{2}$$

where C denotes each criterion, w denotes the weight or priority of each criterion or sub-feature, v denotes the value of a certain feature, SF denotes each sub-feature, m is the number of sub-features for k^{th} criterion, n is the number of criteria, $L_{i,j}$ denotes the i^{th} language for the j^{th} stakeholder, and $C_{i,j,k}$ represents the k^{th} criterion for j^{th} stakeholder and i^{th} language. In Eq. (1), we compute the normalized average weighted score for the k^{th} criterion, j^{th} stakeholder, and i^{th} language. Similarly, the Eq. (2) computes the normalized average weighted score for the i^{th} language and j^{th} stakeholder.

The values of criteria used in Eq. (1) are normal in range [0, 1]. Dividing the weighted sum of the numerator by sum of weights gives also a normalized value in range [0, 1]. This is also the case for Eq. (2). Bounding the values of criteria and intermediate scores in [0, 1] enables us to combine them mathematically. Besides, a final score of, for example, 0.5 shows that the language in question has 50 percent of *desirability* with respect to the considered requirements and preferences.

Farooq et al.[43] proposed a similar score function to evaluate PLs. However, there exist several differences between our score function and that of Farooq et al. For one, Eq. (1) can be used recursively for arbitrary levels of criteria. For two, it is not restricted to a pre-defined set of criteria and has potential to be used with any set of evaluation criteria. For three, the values of each criterion are not restricted to four cases; it can have arbitrary value in range [0, 1]. Finally, applying such a

score function in evaluating of MTLs is novel in addition to employ it in our proposed *decision-support* methodology.

4 Evaluation

This section is organized around a detailed study of applying the proposed evaluation criteria and the proposed methodology on six subject MTLs: ATL, ETL, Kermeta, QVT-O, QVT-R, and TGG. At Section 4.1, we give a brief explanation on required experimental setup. Next, the results of evaluations of subject MTLs on 11 major criteria are presented in Sections 4.2 to 4.12. For each criterion, at first, a detailed description of applying its sub-criteria to evaluate the mentioned languages is given. Then, a table of the summarized results for this criterion is presented to accompany the descriptive results. In Section 4.13, a comprehensive view of the overall results is given. Finally, in Section 4.14, the proposed methodology is evaluated.

4.1 Experimental setup

For experiments, we have installed a number of tools on different Win32 versions of Eclipse. Particularly, we have installed Eclipse Luna 4.4 as a basis for ATL 3.3, Epsilon[3] V1.2, QVT declarative V 0.11, and QVT operational V 3.3. Moreover, Eclipse Kepler 4.3 has been installed as the basis for Kermeta V 2.0.4 and eMoflon[4] V 1.7.0. For EmorF[5] V 0.4.2, Eclipse Juno was applied. Finally, for HenshinTGG[6], the only available version was installed. In order to find the number of keywords and other information for each language, the corresponding reference manuals were used. Moreover, to evaluate QVT-O and QVT-R, the QVT Version 1.2 (OMG) was used.

4.2 Readability

Simplicity: As explained in Section 3.1.1, simplicity is assessed based on basic constructs, primitive features, and minimal operator overloading. We evaluate the number of constructs and features based on abstract syntax for each transformation language. ATL MM contains three packages including ATL components, OCL classes, and primitive types. There are four primitive types in ATL: *String*, *Boolean*, *int*, and *double*. OCL classes include primitive types that are *StringType*, *BooleanType*, *NumericType*, *IntegerType*, *RealType*, to name a few. ETL, however, is specified on top of EOL; so any primitive types are not defined specially for ETL. In EOL, *String*, *Real*, *Integer*, and *Boolean* are defined as primitive types. EOL provides some collection types including *Set*, *OrderedSet*, *Bag*, and *Sequence*[18]. The basic types in Kermeta are divided into three groups: primitives, enumerations, and local data types. Primitives contain *Integer*, *String*, and *Boolean*[12]. QVT languages are defined on top of each other which contain seven MMs. The primitive types defined in QVT are based on Java primitive types that are *Boolean*, *Integer*, *Real*, *String*, *UnlimitedNatural*[13]. For TGG, there is no standard MM, because it has been

[3] `www.eclipse.org/epsilon/`
[4] `http://www.emoflon.org/emoflon/`
[5] `http://emorf.org/index.html`
[6] `http://de-tu-berlin-tfs.github.io/Henshin-Editor/`

developed by several universities; the primitive types defined in TGG-Interpreter[7] are Integer, String, and Boolean[44].

ATL supports overloading with respect to its definition. However, all operators which are defined in the same context need to have distinct names. ETL and QVT-O support operator overloading even within the same context. Overloading is not allowed in Kermeta[35]. In QVT-R, overloading is supported but it does not allow specifying context for a function. Due to the graphical nature of TGG, the semantic of the transformation rule redefinition refers to generalization in which redefined TGG rules extend the original rule[35].

Orthogonality: Orthogonality of a language is assessed based on *parameterization* features and *data type completeness*. Considering the features of parameterization, ATL supports control parameters and higher-order rules because it is possible for ATL to load ATL programs as input models[45]. However, ATL does not support generics. ETL rules cannot provide parameterization features, and these features can be determined in the level of operations[46]. Kermeta can support control parameter and generics such as generic class or generic operations but cannot provide higher-order rules[12]. QVTs, i.e., both relational and operational languages support simple control parameters; however, they cannot support generics. QVT-R does not provide higher-order rules. QVT-O is able to load and transform models conforming to QVT MM, which leads to provide higher-order rules. TGG cannot support control parameters; however, it can provide generic functions and higher-order rules[35]. Considering data type completeness, none of the languages can provide this feature.

Syntax design: As explained in Section 3.1.1, this criterion is determined based on the number of keywords, methods of forming compound statements, and possibility of one-to-one mappings between syntax and semantic. ATL keywords are divided into three groups[8]: constant keywords such as *true* and *false*, type keywords such as *Bag*, or *Set*, and language keywords like *if*, *foreach*, and *uses*. The number of keywords is 45 and 10 for ATL and ETL respectively. To note that ETL takes some keywords from EOL and we do not consider them in our analysis. This decision is made to manifest the difference between ATL and ETL clearer. Some examples of ETL keywords are *rule*, *transform*, *to*, *guard*, and *extends*[18]. As explained before Kermeta is a general purpose language, so it is large and it has 51 keywords such as *setter*, *using*, *until*, *raise*, *package*[12]. Because of the imperative nature of QVT-O, it is a large language with 117 keywords[13]. The samples of QVT-O keywords are *access*, *assert*, *reject*, *refines*, and *typedef*. QVT-R has 15 significant keywords that cannot be used as identifiers such as *top*, *relation*, *when*, *where*, and *enforce*[13]. TGG is a graphical language without any standard definition, so we can assume it has no significant keywords. This decision roots from the graphical nature of TGG which is substantially different from other subject MTLs. Another option is to consider the graphical representations as symbols of that language. However, as we are not investigating a certain TGG language, the former case is used.

In ATL, there exist several ways to define a compound statements. The condition statement in ATL can be specified by *if* which evaluates a condition; this condition

[7] http://www.cs.uni-paderborn.de/index.php?id=tgg-interpreter
[8] https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language#ATL_Keywords

should be defined as an OCL expression. Since it is possible to define some statements in the then clause, ATL can support different ways of forming compound statements such as specifying nested condition or loop statements. It is worth to mention that ATL uses for statements to define loops.

ETL provides EOL different statements such as if and switch for defining conditional statements and for, *foreach*, and *while* for specifying looping statements. Because it is similar to Java PL, it can support several ways for forming compound statements[18]. Kermeta is an object-oriented PL which provides block, conditional, and loop statements by *do*, *if/else*, and *until* keywords[12]. QVT-O can specify block statements by *do/end* blocks, conditional statements by *if/elif/else*, *switch* clauses and loop statements by *foreach*, *forone*, *compute* and *while* expressions[13]. QVT-R and TGG can only uses OCL expressions such as if and let expressions, so they are very limited in this case.

Since all MTLs are defined on top of the OCL language or use it, and there is no one-to-one mapping between syntax and semantic of OCL expressions, they cannot support form and meaning feature. For example, for determining an undefined object, it is possible to use *isUndefined()* operation or *not isDefined()*.

Syntactic separation: Syntactic separation is provided by ATL rules because of its hybrid style. This is achieved by *from/to* keywords in ATL. ETL has also a hybrid paradigm that it can support syntactic separation with the use of *to* keyword. Due to the imperative nature of Kermeta and QVT-O, they do not provide syntactic separation; similar to traditional PLs, in Kermeta, operating parts of source and target domains are mixed. In QVT-O mappings, the left side of an expression is the target element and the right-hand side addresses source elements. However, this language cannot provide clear separation. QVT-R is able to specify rules with clear separation between source and target operating parts because its style is declarative. Similar to QVT-R, TGG is a declarative language which provides clear separation due to the fact that both models have distinct parts. The degree to which TGG supports this feature is higher as compared to other subject MTLs.

Application condition: Except for Kermeta, other subject MTLs support application condition. In ATL programs, it is possible to create Boolean expressions inside *from* block of the rule which are executed if the specified condition becomes true. ETL supports this feature by defining condition in the guard block of the rules. Application condition is provided by QVT languages via defining conditions in the *when* clauses. In TGG rules, one can specify OCL condition to support this feature. Since ATL and ETL support this feature explicitly in the rule, they are stronger than other languages.

Rule scheduling: In ATL, form of scheduling is implicitly supported by matched rules, and it is possible to explicitly invoke some rules from other rules. Similar to ATL, ETL supports this feature as a mix of implicit/explicit scheduling; In the pre/post blocks, user-defined operations, and pre-defined methods such as *satisfies*(), the ETL rules can be explicitly called. Kermeta rule scheduling is specified by developer explicitly. The rule scheduling form in QVT-O is defined explicitly and internally because of its imperative paradigm. In QVT-R, at first all top relations are executed, and then they are followed by the rules specified in the *when* clauses, so the form is determined implicitly. However, it is possible to call a

rule from another one, which enables developer to schedule rules explicitly. TGG uses layering for external rule scheduling[36].

Domain body: Domain body is investigated from variable, pattern, and logic viewpoints. In all languages, meta-variables are considered. The structure of all languages except TGG is term-based. The abstract syntax of each language is specified as a MM. The concrete syntax of all languages except TGG is textual; however, OMG specify a graphical concrete syntax for QVT-R that its tools have not yet implemented graphical concrete[13]. It is worth to mention that ATL, ETL, and QVT-R have clearer textual syntax than Kermeta and QVT-O. Element creation in QVT-O, Kermeta, and TGG is explicitly specified; other languages implicitly create elements. ATL, ETL, QVTs, and Kermeta specify values by imperative assignment, which is more preferable method. TGG specify values by value binding.

Tracing: All of the subject MTLs except Kermeta offer dedicated support for tracing. Because of imperative paradigm of Kermeta, it is possible to implement some structures to support traceability. All subject MTLs, which have dedicated support of traceability, keep trace information in a separate model.

The summary of evaluations for the sub-criteria of readability is demonstrated in Table 2.

Table 2 The results of evaluating subject MTLs on the sub-criteria of readability; ○ (not supported) < ◗ (partially supported) < ● (highly supported)

Sub-criteria	ATL	ETL	Kermeta	QVT-O	QVT-R	TGG
Simplicity (#primitives)	9	0	5	5	5	3
Simplicity (Minimal overloading)	◗	○	●	○	◗	◗
Orthogonality (Control parameters)	●	○	●	●	●	○
Orthogonality (Generics)	○	○	●	○	○	●
Orthogonality (Higher-order rules)	●	○	○	●	●	●
Orthogonality (Data type completeness)	○	○	○	○	○	○
Syntax design (#Keywords)	45	10	51	117	15	0
Syntax design (Compound statements)	●	●	●	●	○	○
Syntax design (Syntax and semantic)	○	○	○	○	○	○
Syntactic separation	◗	◗	○	○	◗	●
Application condition	●	●	○	◗	◗	◗
Rule scheduling	◗	◗	●	●	◗	●
Domain body (Variable)	●	●	●	●	●	●
Domain body (pattern-structure)	Term	Term	Term	Term	Term	Graph
Domain body (pattern-abstract syntax)	●	●	●	●	●	○
Domain body (Pattern-concrete syntax)	◗	◗	○	○	◗	●
Domain body (Logic-value specification)	●	●	●	●	●	◗
Domain body (Logic-element creation)	Implicit ○	Implicit ○	Explicit ●	Explicit ●	Implicit ○	Explicit ●
Tracing (dedicated support)	●	●	○	●	●	●
Tracing (Separate model)	●	●	○	●	●	●

4.3 Writability

The evaluation of subject MTLs for writability sub-criteria is demonstrated in the course of this Section. Note that some of the sub-criteria are explained in the

Section 4.2.

Simplicity and orthogonality: These two criteria are assessed based on several features. In this Section, we only show the results of evaluation by comparing the subject MTLs. From the number of primitive's point of view, all subject MTLs provide acceptable number of primitives. Among them ETL is the simplest which only use EOL structures and does not define new primitive types. In contrast, ATL defines 9 different primitive types. From the orthogonality point of view, all MTLs except ETL can be considered in the same situation; however, ETL is not an orthogonal MTL.

Syntax design: As explained before, ATL, Kermeta, and specially QVT-O are large languages with several keywords. However, ETL and QVT-R are smaller which defines small number of keywords and uses the keywords of their host languages. Since TGG is not a textual language, it does not use any words to define transformation rules. Considering compound statements all subject MTLs except declarative ones provides several ways to form compound statements which lead to high expressivity and simple writability.

Abstraction: Abstraction is determined based on supporting specification of abstract rules, operation overloading, simplification, selection, generalization, reflection, aspect, and modularity.

Concerning abstract rules, all subject MTLs except QVT-R support this type of abstraction[35]. Regarding operation overloading, ETL and QVT-O provide complete support. However, ATL and QVT-R provide limited support to this feature.

QVT-R and TGG support simplification by abstracting from control flow[36]. ATL provides this feature with the use of helper functions and attributes. ETL also is the same as ATL which can abstract from the control flow by the means of its declarative part of style. However, Kermeta and QVT-O cannot support simplification because of their imperative syntax.

Due to the existence of selection feature, TGG can define correspondence nodes on a certain part of MMs[36]. However, other languages cannot support this feature. Concerning generalization, Kermeta can provide generic types to support generalization and the eMoflon implementation of TGG has also provided genericity[36].

Reflective access to transformation is allowed by ATL during execution[33]. ETL inherits reflection from its host language. The reason why Kermeta is able to support reflection can be attributed to its special style, which is known as an aspect-oriented language[12]. QVT-O provides reflection support that offers to access trace information[16]. Reflection is explicitly supported by TGG with the use of Story Driven Modeling[35].

From the modularity point of view, ATL can keep frequently used helpers in libraries to reuse them in other modules. However, ATL modules cannot be reused in other modules. Hence, in ATL, only using helper functions can lead to modularity[10]. In ETL, it is possible to organize rules in one module and import that module in another one[46]. Kermeta offers to organize transformations into classes that it is possible to split a transformation into different files[12]. In QVT-O, transformations can be reused by access and extend mechanism which are similar to import facility of PLs and inheritance property of object-orientation respectively[35]. QVT-R supports

modularity by importing or extending QVT transformation and using its rules in *when* or *where* clauses[35]. TGG can merge the high-level rule type from one transformation with that low-level type in a new one[35]. The results of evaluation of abstraction are summarized in Table 3.

Expressiveness: Expressiveness is defined based on Turing-Completeness and having the appropriate and natural ways to define algorithms. As mentioned, ATL and ETL are multi-paradigms, or hybrid, which involve imperative paradigm as well. In imperative paradigm, there are appropriate ways to explain how to express algorithms. They are general rule-based languages which are Turing-complete. ATL uses global helpers or lazy rules to support recursion; looping in ATL needs to have bounded number of steps. ATL is a unidirectional language which has low expressive power in bidirectional applications. ETL is defined on top of EOL, which support unbounded looping and recursion. Kermeta and QVT-O are imperative languages that provide Turing-Completeness, and unbounded recursion. For the case of QVT-O, unbounded looping is not supported. These two imperative languages cannot be applied in bidirectional case studies. QVT-R is a bidirectional rule-based language which has declarative style; it is possible to call QVT operational mappings from QVT-R transformations that result in Turing-Completeness. It can support unbounded recursion but does not provide looping.

Table 3 The results of evaluating subject MTLs on abstraction criteria; ○ (not supported) < ◗ (partially supported) < ● (highly supported)

Abstraction sub-features	ATL	ETL	Kermeta	QVT-O	QVT-R	TGG
Abstract rules	●	●	●	●	○	●
Operator overloading	◗	●	○	●	◗	◗
Simplification	●	●	○	○	●	●
Selection	○	○	○	○	○	●
Generalization	○	●	●	●	○	◗
Reflection	◗	●	●	◗	○	●
Aspect	○	○	●	○	○	○
Modularity	◗	●	●	●	●	◗

For TGGs it turns out that there are challenges to their expressive power[14]. Expressive power is one of the four design principles that a useful TGG should not violate. To our knowledge, none of the TGG approaches and the corresponding algorithms satisfies all four properties. Often in practice, the designer had to trade one property for another.

Concerning incrementality, ATL and ETL cannot provide any support. Kermeta and QVT-O do not also offer incrementality by default. However, because of their imperative nature, it is possible to load the existing target model and update as desired which leads to target-incrementality. There is no any way to implement source-incrementality by these imperative languages. QVT-R can support target-incrementality to provide change propagation, but it cannot preserve user updates in the target. It does not offer source-incrementality. TGG inherently support incrementality in both directions, but some of its tools like EmorF do not implement incrementality in their transformation engines[30]. Table 4 (upper part) denotes the summary of evaluation for expressiveness.

Table 4 The results of evaluating subject MTLs on expressiveness and tracing; ○ (not supported) < ◗ (partially supported) < ● (highly supported)

Expressiveness sub features	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
Turing completeness	●	●	●	●	●	○
Natural for applications	◗	●	●	◗	◗	○
Bidirectionality	○	○	○	○	●	●
Incrementality	○	○	◗	◗	◗	●
Tracing sub feature	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
Dedicated support	●	●	○	●	●	●
Automatic creation	●	◗	○	●	●	●

Syntactic separation and application condition: TGG can completely separate the source operating part from the target. After TGG, QVT-R, ETL, and ATL support this feature in the second place. ATL and ETL support application condition explicitly, which has some effects on the simplicity of the writability. The imperative languages, i.e., QVT-O and Kermeta cannot support syntactic separation and application condition.

Rule scheduling: As explained before, when the scheduling of the rules is determined implicitly, the language can be more writable. The two hybrid languages and QVT-R can schedule the rules implicitly. However, Kermeta and QVT-O need to specify scheduling explicitly and TGG schedules rules externally.

Domain body: Writability improves if element creation is carried out implicitly. Hence, ATL, ETL, and QVT-R would be more writable with respect to this point. Readability was described in Section 4.2.

Tracing: If the tracing information can be generated automatically without using any piece of code, the language will be more writable. As a consequence, ATL, QVTs, and TGG can be considered more writable from the trace creation point of view. ETL offers a trace MM, and some statements to support generation of trace models, but Kermeta does not provide any facilities for tracing. Table 4 (lower part) show the summary of results.

4.4 Reliability

In this Section, the evaluation of these sub-criteria is determined for six subject MTLs.

Readability and writability: Since reliability is dependent upon two major criteria, readability and writability, we introduce majority support ranking strategy.

Majority support ranking strategy (MSR): In this strategy, the sub-criteria of the concerning criteria (e.g. readability) are listed. Then, we specify, for each subject MTL, whether it completely supports each sub-criterion (support) or it does not provide that feature at all (not support). Next, we count the number of support situations and not support cases for each language. The languages are sorted in ascending order based on the number of support situations. In case of a tie in the number of support situations for two languages, they are analyzed based on the number of not support cases.

The result of applying MSR strategy on readability is depicted in Table 5. For application condition, ATL and ETL are considered as support and Kermeta as not

support because the first two MTLs support application condition completely and Kermeta does not provide at all. Other subject MTLs are neither support nor not support. At this point, we can measure the subject MTLs from readability point of view: QVT-R = 7 < ETL = 8 < ATL, Kermeta = 9 < TGG = 10 < QVT-O. Due to the MSR strategy, QVT-O gains first place. The second and third places go to TGG and ATL, respectively. ETL and Kermeta gain the fourth place with Kermeta to be superior to ETL due to its smaller losing value. QVT-R gains the last place.

Table 5 The results of evaluating subject MTLs on readability and writability for reliability using MSR strategy

Feature of Readability	**Support**	**Not support**
Simplicity (#primitives)	ETL, TGG	ATL
Simplicity (Minimal overloading)	Kermeta	ETL, QVT-O
Orthogonality (Control parameters)	ATL, Kermeta, QVT-O, QVT-R	ETL, TGG
Orthogonality (Generics)	Kermeta, TGG	ATL, ETL, QVT-O, QVT-R
Orthogonality (Higher-order rules)	ATL, QVT-O, QVT-R, TGG	ETL, Kermeta
Orthogonality (Data type completeness)	-	All languages
Syntax design (#Keywords)	QVT-O	TGG
Syntax design (Compound statements)	ATL, ETL, Kermeta, QVT-O	QVT-R, TGG
Syntax design (Syntax and semantic)	-	All languages
Syntactic separation	TGG	Kermeta, QVT-O
Application condition	ATL, ETL	Kermeta
Rule scheduling	Kermeta, QVT-O, TGG	-
Domain body (Variable)	All languages	-
Domain body (Pattern-structure)	-	-
Domain body (Pattern-abstract syntax)	ATL, ETL, Kermeta, QVT-O, QVT-R	TGG
Domain body (Pattern-concrete syntax)	TGG	Kermeta, QVT-O
Domain body (Logic-value specification)	ATL, ETL, Kermeta, QVT-O, QVT-R	-
Domain body (Logic-element creation)	Kermeta, QVT-O, TGG	-
Tracing (Dedicated support)	ATL, ETL, QVT-O, QVT-R, TGG	Kermeta
Tracing (Separate model)	ATL, ETL, QVT-O, QVT-R, TGG	-

Analysis (readability)	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
#Support cases	9	8	9	11	7	10
#Not support cases	4	5	7	6	4	6
Analysis (writability)	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
#Support cases	14	17	14	14	14	14
#Not support cases	9	10	14	12	8	11

Concerning the writability criterion, the same process of the proposed strategy is repeated. After measurement the supports and not supports cases for each language, the results are demonstrated in Table 5 (bottom). By sorting the languages the following result is obtained: ATL, Kermeta, QVT-O, TGG, QVT-R = 14 < ETL = 17. It turns out that ETL gains the first place. However, for the rest of the languages that the number of their support is equal, the number of losing cases is analyzed. Since the losing value of QVT-R is smaller, it achieves the second place. ATL, TGG, QVT-O, and Kermeta achieve the third, fourth, fifth, and sixth place, respectively.

Rule application strategy: ATL and ETL provide location determination in a

deterministic way[10,45]. In Kermeta terminology, rules are interpreted as operations and developers must decide which operation is executed at specific point; therefore determining the value for Kermeta does not make sense[45].

In QVT-O, there is only one entry point for each process, so its strategy is deterministic[46]. QVT-R also offers a deterministic strategy; in the first step all top rules are executed and their order is determined based on the defined *when* clauses. TGG can provide a deterministic strategy, which is performed based on the traceability links.

Exception handling: ATL, ETL, and imperative languages, i.e., Kermeta and QVT-O can handle exceptional transformations that need to be specified explicitly by the user. Since ETL is able to use EOL statements, it is possible to throw exceptions like Java exceptions[18]. Kermeta can support exception handling like other object-oriented PLs[12]. QVT-O can evaluate a condition and throw exceptions[23]. QVT-R medini tool can specify a run-time exception handling that prints the message of error with the number of erroneous line[17]. TGG cannot support exception handling.

Aliasing: Aliases can occur in ATL, ETL, Kermeta, and QVT-O due to their imperative paradigm, however, TGG and QVT-R cannot provide aliasing. Therefore, these two declarative languages can be more reliable.

Domain typing: All languages support domain typing syntactically. ATL compiler does not check typing in the compile time, and all the type checking is performed dynamically[47]. ETL compiler behaves the same as ATL. However, Kermeta is a strongly typed language in which all types are checked at compile time and the syntax is annotated with information of type[47]. QVT languages provide weak type checking statically. Some parts of these languages that are defined based on the OCL language are strongly typed, because OCL is strongly typed[48]. TGG provides static type checking but it is not strongly typed[35]. The overall results of reliability are outlined in Table 6.

Table 6 The overall results of evaluating subject MTLs for reliability; ○ (not supported) < ◗ (partially supported) < ● (highly supported)

Sub Criteria	ATL	ETL	Kermeta	QVT-O	QVT-R	TGG
Readability	QVT-R < ETL < Kermeta < ATL < TGG < QVT-O					
Writability	Kermeta < QVT-O < TGG < ATL < QVT-R < ETL					
Rule strategy	●	●	◗	●	●	●
Exception handling	●	●	●	●	○	○
Aliasing	Yes	Yes	Yes	Yes	No	No
Domain typing	○	○	●	◗	◗	◗

4.5 Maintainability

Maintainability is determined based on readability, size of transformation, its complexity, and abstraction. The size of transformation is assessed based on lines of transformation code.

The complexity is evaluated by the total number of occurrences of operators, features, operations and rule calls. To compute the size and complexity, we consider the example of class diagram to database schema transformation[49], and we simplify it to only three rules which are package to schema, class to table, and single-valued

attribute to column. The solution of the case study can be found from the published specifications for subject MTLs: ATL[50], ETL[18], Kermeta[12], QVT-O[13], QVT-R[13], and TGG[51].

Readability and abstraction: Readability has a direct effect on the maintainability. The MSR strategy can be used to determine which languages have high readability leading to high maintainability. Applying the proposed strategy on readability studied in measuring Section 4.4: QVT-R < ETL < Kermeta < ATL < TGG < QVT-O.

Abstraction has been completely discussed in Section 4.3. It has explained that all subject MTLs except QVT-R support definition of abstract rules. Abstraction by simplification is provided by two hybrid languages and two declarative ones. Only TGG can provide selection. ATL and QVT-R cannot offer generalization; however, the rest of the languages provide it. Kermeta supports reflection, aspect, and modularity. Modularity is also provided by ETL and QVT languages completely.

Size: The lines of code (LOC) is not a precise feature, however, it is possible to have a rapid analysis for computing the size. The size computation is demonstrated in Table 7 (upper part). Due to the graphical nature of TGG, lines-of-code is not a proper measure to take. Instead, we compute the number of object creation, and OCL constraints or assignments. As the Table 7 shows, ATL, ETL, and QVT-O have small (reasonable) sizes, TGG is larger with a medium size, and Kermeta and QVT-R are large with major difference in sizes.

Complexity: Complexity is assessed by the number of operators, features, operations, and rule calls. The obtained results of investigating the complexity are shown in Table 7 (lower part). Overall, the results suggest that ATL and ETL are less complex than QVT-O, QVT-R, and TGG. Kermeta is the most complex one.

Table 7 The results of evaluating subject MTLs on (size, complexity) criteria; ● (small, low) < ◗ (medium, mediocre) < ○ (large, high)

Size	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
Package to schema	6	4	14	4	9	7
Class to table	6	6	14	7	18	9
Attribute to column	7	7	12	5	23	9
Total	19	17	40	16	50	25
Analysis	●	●	○	●	○	◗
Complexity	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
#Operators	8	8	34	16	22	19
#Features	14	12	13	11	18	12
#Calls	1	1	3	3	4	0
Total	23	21	50	30	44	31
Analysis	●	●	○	◗	◗	◗

4.6 Learnability

As it has been stated, learnability is assessed by three features: learning curve, language size, and maturity.

Learning curve: To analyze learning curve, it is required to determine who wants to work with the language. From the object-oriented programmer point of

view, Kermeta is easier to learn because it uses the syntax of Eiffel languages and is also Java-like. OCL-like syntax makes ATL, QVT-O, and QVT-R more preferable.

ATL is superior to the other two languages, because it utilizes the expressiveness of QVT-O resultant from its imperative-style along with the simplicity of QVT-R which is due to its declarative-style. ETL is a Java-like language defined on top of EOL which is itself OCL-like. In fact, among the mentioned languages, ETL has a low learning curve that leads to high learnability.

TGG requires learning and working with specific formalism. Declarative languages determine what should be mapped by the transformation specification, and they do not provide the control flow to specify how the transformation should be executed. Therefore, the cost of training of these languages will increase.

Language size: The language size is evaluated based on the subject MTLs' MMs. To this end, number of classes, properties, and associations are computed.

Number of keywords is considered as another feature and is demonstrated in Table 8. ATL MM contains three packages including ATL classes, OCL, and primitive types. ATL package contains nine classes, and OCL package includes 56 classes. The most important ATL classes which are related to transformation definition are *LocatedElement*, *Rule*, and *VariableDeclaration*. The classes of OCL packages are related to different kinds of expressions such as *OCL*, *numeric*, *primitive*, *collection*, *loop*, *property call*, and *operation call expressions*. ETL MM is defined on top of EOL MM that it contains only eight classes *ETLModule*, *NamedBlock*, *TransformationRule*, and *Guard* to name a few. These classes use six EOL classes which lead to a total of 14 classes in ETL[18]. QVT languages are defined on top of each other which contain seven MMs[13]. QVT-O MM is defined based on six other packages including *EMOF*, *ImperativeOCL*, *EssentialOCL, QVTBase, QVTRelation,* and *PrimitiveTypes*. QVT-O, itself, contains 22 classes, which are only related to the transformation components and uses 18 classes of other MMs. QVT-R MM is specified based on three other MMs named *EMOF*, *EssentialOCL*, and *QVT-base*. The MM, itself, contains only nine classes, and uses 15 classes of aforementioned packages. There is no standard MM for TGG, and each TGG tool use its own MM. To compute the language size, we consider the Henshin[9] TGG MM.

Table 8 The results of evaluating subject MTLs on language size; ● (small) < ◖ (medium) < ◗ (large) < ○ (very large)

Language size	ATL	ETL	Kermeta	QVT-O	QVT-R	TGG
#Classes	65	14	32	40	24	31
#Properties	24	0	11	11	1	105
#Associations	81	17	25	41	20	46
#Keywords	45	10	51	117	15	0
Overall	215	41	119	219	60	182
Analysis	○	●	◖	○	●	◗

Maturity: The first version of ATL was released in 2003[52]. Then, it was followed as an Eclipse project in 2006. There exist more than 100 industrial and academic case studies which are implemented by ATL. Due to long history, deeper

[9] `http://de-tu-berlin-tfs.github.io/Henshin-Editor/`

experience in terms of available formal case studies[73], and consequently higher maturity, ATL can be considered as baseline for comparison. The ATL forum[10] is still active and has been separated since 2012; before that there was one single forum for all M2M transformation Eclipse projects. In ATL forum, there exist more than 2390 messages. ETL has been available since 2008[20] yet there is not large number of transformation case studies by this language. The Epsilon forum[11] has been active since 2011 with more than 7180 messages. The first version of Kermeta has been reported since 2005 until 2012 with more than forty case studies[17]. Kermeta forum[12] with 413 messages has not been active since 2014.

For the case of QVT languages, several tools have implemented these languages. SmartQVT is an inactive QVT operational tool that its last release happened in 2008. Eclipse QVT-OML was migrated from GTM/UMLX project in 2008 and its last version is released in 2015, which shows its activeness. QVT-OML forum[13] has been activated since 2012 with more than 780 messages. Medini is another QVT tool which has implemented QVT-R between 2007 and 2011. It has twenty published case studies[17]. Eclipse QVT-Declarative has been in the incubation phase since 2012 but its forum has been activated since 2012.

TGG has six tools: eMoflon (2012 to now) with issue tracker page and available contact email, TGG-Interpreter (2006–2011) with contact email, HenshinTGG (2012 to now, 2015) with issue page and several case studies, MoTE[14] (2007–2012) without any support, ATOM3[15] (until 2002) without any support, and EMorF (2011 to 2012) with contact email. So the number of years is estimated between three to five with several case studies, and active support for at least two tools.

The number of years for history can be divided into three groups: less than four years, four to eight years, and more than eight years[17]. The results of three features are shown in Table 9. As a consequence, ATL and ETL are the most learnable languages among other languages. After that, QVT-O and Kermeta have reasonable values of learnability. Finally, learning QVT-R and TGG spends much time and cost.

Table 9 The results of evaluating subject MTLs on maturity

Maturity	ATL	ETL	Kermeta	QVT-O	QVT-R	TGG
History of use	12 years	7 years	7 years	9 years	4 years	5 years
Support for case studies	High	Low	Medium	Medium	Low	Medium
Technical support	High	High	Low	High	Low	Medium

4.7 Generality

All subject transformation languages offer exogenous model transformations as well as support for M-to-N cardinality transformations and all technical spaces including EMF, XMI, and XML. As stated, ATL provides dedicated support for traceability but it does not offer multidirectionality and change propagation[53].

[10] https://www.eclipse.org/forums/index.php/f/241/
[11] https://www.eclipse.org/forums/index.php/f/22/
[12] https://gforge.inria.fr/forum/?group_id=32
[13] https://www.eclipse.org/forums/index.php/f/244/
[14] https://www.hpi.uni-potsdam.de/giese/public/mdelab/mdelab-projects/mote-a-tgg-based-model transformation-engine/
[15] http://atom3.cs.mcgill.ca/

Among four operations, Create, Read, Update, and Delete (CRUD), it can offer the first three ones. ATL transformations can be defined and executed by two modes[10]: normal or refining mode. Refining mode allows endogenous transformations such as refactoring and refinement. Model comparing is not supported directly by ATL, but Atlas Model Weaver[54], which can be integrated with ATL, supports model comparing, weaving, and merging. AMW has not been updated since 2008; hence, it cannot be used with new releases of Eclipse. Since ATL is defined based on the OCL language, it inherently offers model validation. Model-to-text transformations cannot be implemented by ATL, but it can be integrated with Acceleo, that both projects are developed by obeo[16] model-driven company.

ETL offers traceability, but not bidirectionality and change propagation. It only supports Read and Create operations. It does not provide any other features but can be integrated with other Epsilon languages including Epsilon Wizard Language (EWL) for in-place updates, Epsilon Merging Language (EML) for model merging, Epsilon Comparison Language (ECL) for model comparing, Epsilon Validation Language (EVL) for validating, and Epsilon Generation Language for code generating.

Kermeta does not offer facilities for traceability and multidirectionality. Since it is target-incremental, it can provide change propagation to the target side. Moreover, it supports all of the CRUD operations in addition to providing in-place updates including refactoring and refinement. However, Kermeta cannot support model merging. Further, no plug-in or tool exists in Kermeta Model Development Kit[17] (MDK) that can integrate with Kermeta. Additionally, there is no context about the ability of Kermeta in model comparing, but because of its style it seems this feature is allowed. Finally, Kermeta offers model checking in its beta version to support model validation that can be integrated with Kermeta Emitter Template[18] (KET) and template-based text generator to support code generation.

QVT-O provides traceability and propagating changes to the target side, and complete support of CRUD. Although it cannot support endogenous transformations[55], but model merging is offered[13]. Since it is specified based on OCL components, model validation is allowed by QVT-O. QVT-R supports bidirectionality, dedicated traceability, change propagation to both sides of transformation, complete CRUD operations, in-place transformations[13], model comparing, and merging[56]. Because QVT-R transformations perform consistency checking before enforcement, model validation is supported by this language. TGG provides bidirectional transformations, trace information, model synchronization leading to change propagation, complete support for CRUD operations, and in-place transformation[19]. Model merging and comparing could be possible by TGG[57]; however, no effective solution has been implemented yet. Model validation can be performed by story diagrams; MoTE TGG tool can generate story diagrams from TGG rules leading to model validation. eMoflon proposes a solution for model-to-text transformation. Table 10 summarizes the results. It turns out that QVT-R and TGG are the most general languages. Then, QVT-O and Kermeta gain

[16] `http://www.obeo.fr/`
[17] `http://www.kermeta.org/mdk/`
[18] `http://www.kermeta.org/mdk/ket`
[19] `www.emorf.org`

the third and fourth positions, respectively. Next, the fifth place goes to ATL. ETL is specifically for out-place exogenous M2M unidirectional transformations. The features of generality are supported by Epsilon family languages, therefore stand-alone ETL cannot provide generality.

Table 10 The results of evaluating subject MTLs on generality

Generality	ATL	ETL	Kermeta	QVT-O	QVT-R	TGG
Source cardinality	Many	Many	Many	Many	Many	Many
Target cardinality	Many	Many	Many	Many	Many	Many
Technical Space (EMF/XMI/XML)	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓
Bidirectionality	✗	✗	✗	✗	✓	✓
Traceability	✓	✓	✗	✓	✓	✓
Change propagation	✗	✗	✓	✓	✓	✓
CRUD transformations	✓✓✓✗	✓✓✗✗	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Endogenous transformation	✓	EWL	✓	✗	✓	✓
Exogenous transformation	✓	✓	✓	✓	✓	✓
Model merging	AMW (old releases)	EML	✗	✓	✓	✗
Model comparing	AMW	ECL	✓	✓	✓	✗
Model validating	✓	EVL	✓	✓	✓	✓
Integrated to code generator	Acceleo	EGL	KET	Acceleo	✗	eMoflon

4.8 Portability

According to what was stated in background, there are several tools for each subject MTL. However, we only consider those tools that can be installed as plug-ins on Eclipse IDE. Eclipse can be executed on different platforms. As such, the subject tool (plug-in) can be portable on several platforms (hardware or operating system). Due to the diversity of Eclipse versions, we consider the last six versions including Helios (H), Juno (J), Indigo (I), Kepler (K), Luna (L), and Mars (M). Then, for each transformation tool, we check whether it can be installed on each version. ATL and QVT-OML tools are implemented as the Eclipse MMT project; so they are available as soon as Eclipse version is released. The last version of Epsilon based on Mars has not been released yet (at the time of writing this paper). Kermeta has not been updated after Kepler and QVT-R after Indigo. TGG has been followed through six tools that they have not been active for a long time. For example, AToM3 cannot be installed on these versions anymore. MoTE and TGG-Interpreter can be installed on only Helios version. The Juno version supports EMorF, and HenshinTGG is provided on Luna. Table 11 summarizes the results.

Table 11 The results of evaluating subject MTLs on portability

Portability	ATL	ETL	Kermeta	QVT-O	QVT-R	TGG
Hardware (32/64bit)	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
OS (Win/Linux)	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
Version (H/J/I/K/L/M)	✓✓✓✓✓✓	✓✓✓✓✓✓✗	✓✓✓✓✗✗	✓✓✓✓✓✓	✓✓✓✗✗✗	Only one[1]

[1]Each of the six tools can be installed on just one IDE

4.9 Reusability

Reusability is affected by abstraction, rule inheritance, higher-order rules, and composition. ATL can only support single inheritance. ETL, Kermeta, QVT-O, and TGG provide multiple inheritance[8]. However, QVT-R cannot support inheritance[45]. Among the subject MTLs, ATL, QVT-O, and TGG support higher-order rules[35]. To support composition, ATL and ETL provide lazy rules. ETL also specifies *satisfies*, and *satisfiesAll* functions to compose transformation rules and uses *pre/post* blocks. Composition specification in Kermeta and QVT-O is explicitly performed by the means of imperative statements. In QVT-R, composition is possible with the use of *when* and *where* clauses. In ATL, ETL, and QVT-R if there is no explicit use of composition, it is performed implicitly. TGG uses layering to specify composition externally[36]. The results for reusability are depicted in Table 12.

Table 12 The results of evaluating subject MTLs on reusability; ○ (low) < ◖ (medium) < ◗ (high) < ● (very high)

Abstraction sub-features	ATL	ETL	Kermeta	QVT-O	QVT-R	TGG
Abstraction (Abstract rules)	●	●	●	●	○	●
Abstraction (Overloading)	◗	●	○	●	◗	◗
Abstraction (Simplification)	●	●	○	○	●	●
Abstraction (Selection)	○	○	○	○	○	●
Abstraction (Generalization)	○	●	●	●	○	◗
Abstraction (Reflection)	◗	●	●	◗	○	●
Abstraction (Aspect)	○	○	●	○	○	○
Abstraction (Modularity)	◗	●	●	●	●	◗
Rule inheritance	◗	●	●	●	○	●
Higher-order rules	●	○	○	●	●	●
Composition	◖	◖	◗	◗	◖	●

4.10 Availability of tools

ATL IDE contains editor, compiler, and debugger. The transformation engine of ATL must compile the ATL code into a byte code that is executed by the ATL virtual machine (EMF VM). An outline view, syntax highlighting, and error reporting are provided by ATL editor. ATL IDE plug-in is available in the Eclipse Modeling Tool, which is needed for all model transformation Eclipse plug-ins. ATL is developed by Obeo Company as the MMT Eclipse project, which can be considered as a sponsorship. Three versions of this tool have been released since 2013.

ETL is one of the proposed languages by Epsilon. The Epsilon tools, Exeed editor, ModeLink (weaver), and Workflow are only some of the tools that are developed besides Epsilon languages. ETL is able to use Java statements and can be used in Java programs. Epsilon does not offer a debugger for ETL or other family of languages. It is necessary to install GMF and Emfatic projects before Epsilon installation[20]. Therefore, the number of Epsilon requirements is three (GMF, Emfatic, and Epsilon). Three versions have been released since 2013.

[20] `https://www.eclipse.org/epsilon/download/`

Kermeta provides an editor with syntax highlighting and auto-completion, and a perfect debugger. It can be easily installed without any requirements on Eclipse. However, there is no sponsorship for Kermeta and it has not been developed since 2012. QVT-OML plug-in tool is available in the Eclipse Modeling tool. Its editor provides syntax highlighting, an outline view, and error detection. There is no debugger for QVT-OML. The language is supported by OMG and developed by Willink Transformations Ltd.[21] There are four release versions for QVT-O. QVT-Declarative is also supported by OMG and developed by Ed. Willink in the Eclipse Modeling tool. However, it is in the incubation phase and there is no compiler or interpreter for it. The newly TGG tool is HenshinTGG which is available for Eclipse Luna (4.4). It is required to install Eclipse Modeling Tool, GIT plug-in, and Zest[22]. It is needed to perform a number of settings. There is no sponsorship for HenshinTGG. The tool provides user-friendly editor with model validating. It provides a debugging process for single forward rules[58]. The summarized results have been shown in Table 13.

Table 13 The results of evaluating subject MTLs on availability of tools; ○ (low) < ◗ (medium) < ● (high)

Availability of Tools	ATL	ETL	Kermeta	QVT-O	QVT-R	TGG
Translator	●	●	●	●	○	●
Editor	●	●	●	●	●	●
Debugger	●	○	●	○	○	●
Easy installation	1	3	1	1	1	4
History of use	12 years	7 years	7 years	9 years	4 years	5 years
Support for case studies	High	Low	Medium	Medium	Low	Medium
Technical support	High	High	Low	High	Low	Medium

4.11 Standardization

Due to the timeliness feature of standardization, QVT languages are early-standardized. OMG proposes QVTs since 2001. ATL is a QVT-like syntax language that has been proposed as an answer to the OMG QVT Request for Proposals (RFP); it is timely-standardized. ETL and Kermeta are timely-standardized because their abstract syntax MMs have been standardized after few years of language design. TGG is lately-standardized for model-driven context because it was firstly proposed by Andy Schürr in 1994[59]. Concerning conformance, only QVT-R translator cannot conform to the whole language standard[25,60]. ATL, ETL, QVT-O are the languages that are upgraded due to Eclipse development standards. However, for QVT-R, there is still no effective Eclipse translator to be upgraded and other implementations such as ModelMorf and Medini are outdated. Kermeta has not upgraded since 2012, notwithstanding Eclipse modeling tools and frameworks have been upgraded since 2014. The aforementioned TGG tools either have not been upgraded or have been developed newly. Table 14 shows the results.

[21] http://www.edwillink.plus.com/willinktransformations/index.html
[22] https://github.com/de-tu-berlin-tfs/Henshin-Editor/wiki/Install_HenshinTGG

Table 14 The results of evaluating standardization of subject MTLs

Standardization	ATL	ETL	Kermeta	QVT-O	QVT-R	TGG
Timeliness	Timely	Timely	Timely	Early	Early	Lately
Conformance	Yes	Yes	Yes	Yes	No	Yes
Obsolescence	Upgraded	Upgraded	Outdated	Upgraded	Outdated	Outdated

4.12 Cost

Since the cost criterion is related to all of the mentioned criteria, we first apply the MSR strategy (see Section 4.4) on all subject MTLs based on the aforementioned criteria including readability, writability, reliability, maintainability, learnability, generality, portability, reusability, availability of tools, and standardization. This leads to establish a separate ranking with at most six places among subject MTLs with respect to each criterion. The results are presented in Table 15. In the next stage, six costs were defined in increasing order according to six ranking places. A cost of +1 is the lowest cost for the language with highest rank. A cost of +6, which is the highest cost, goes to the language with lowest rank. For example, the cost of ATL for readability is +3 since it is the third place. Similarly, the cost of TGG is +5 with respect to learnability. At this point, the costs of all subject MTLs with respect to each major criterion have been determined.

Table 15 The results of evaluating subject MTLs on cost criterion using MSR strategy

Criteria	1st place (cost: +1)	2nd place (cost: +2)	3rd place (cost: +3)	4th place (cost: +4)	5th place (cost: +5)	6th place (cost: +6)
Readability	QVT-O	TGG	ATL	Kermeta	ETL	QVT-R
Writability	ETL	QVT-R	ATL	TGG	QVT-O	Kermeta
Reliability	QVT-O	ETL	TGG	ATL	Kermeta	QVT-R
Maintainability	ETL	QVT-O	Kermeta	TGG	ATL	QVT-R
Learnability	ATL, ETL	QVT-O	Kermeta	QVT-R	TGG	-
Generality	QVT-R	TGG	QVT-O	Kermeta	ATL	ETL
Portability	ATL, QVT-O	ETL	Kermeta	QVT-R	TGG	-
Reusability	TGG	ETL	QVT-O	Kermeta	QVT-R	ATL
Availability of Tool	ATL	QVT-O	Kermeta	TGG	ETL	QVT-R
Standardization	ATL, ETL	QVT-O	Kermeta	TGG	QVT-R	-

In order to determine the total cost of each subject MTL with respect to a certain type of cost, the obtained costs of Table 15 are used. The procedure is as follows. At first, the influencing criteria on each type of cost are determined. Then, for each determining criterion, the cost of each subject MTL is taken from Table 15 with respect to that criterion. Next, the results are summarized in a table. Finally, the total cost of each subject MTL with respect to a certain type of cost is achieved. In the following, the results of computations for each certain type of cost are presented.

Cost of training: Cost of training is related to readability and learnability. The upper part of Table 16 presents how this type of cost can be computed. The readability and learnability cost of each language are computed based on the values in Table 15 The total cost of training is achieved by adding the cost of mentioned

criteria for each language. Due to the results of Table 16 (upper part), QVT-O is the language with lower cost of training and QVT-R has the high cost.

Table 16 The results of evluating subject MTLs on cost of training (upper) and cost of writing a program (lower)

Cost of training	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
Readability cost	+3	+5	+4	+1	+6	+2
Learnability cost	+1	+1	+3	+2	+4	+5
Total cost	+4	+6	+7	+3	+10	+7
Cost of writing programs	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
Readability cost	+3	+5	+4	+1	+6	+2
Writability cost	+3	+1	+6	+5	+2	+4
Reliability cost	+4	+2	+5	+1	+6	+3
Reusability cost	+6	+2	+4	+3	+5	+1
Availability of tool cost	+1	+5	+3	+2	+6	+4
Total cost	+17	+15	+22	+12	+25	+14

Cost of writing a program: The cost of writing a program is affected by readability, writability, reliability, reusability, and availability of tools. The cost of writing is computed in Table 16 (lower part). According to the results for the cost of writing programs, QVT-O and subsequently TGG have low costs. ETL and consequently ATL have medium costs. However, QVT-R and Kermeta have a very high cost.

Cost of compiling and executing: This type of cost is influenced by writability, reliability, and portability. Table 17 (upper part) measures the total cost of compiling and executing. ETL is considered as an efficient language in compiling and executing. After that, QVT-O and ATL go to the second and third place. Finally, TGG and QVT-R are better than Kermeta.

Table 17 The results of evaluating subject programs on cost of compiling (upper) and implementation (lower)

Cost of compiling and executing	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
Writability cost	+3	+1	+6	+5	+2	+4
Reliability cost	+4	+2	+5	+1	+6	+3
Portability	+1	+2	+3	+1	+4	+5
Total cost	+8	+5	+14	+7	+12	+12
Cost of implementation system	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
Readability cost	+3	+5	+4	+1	+6	+2
Writability cost	+3	+1	+6	+5	+2	+4
Reliability cost	+4	+2	+5	+1	+6	+3
Availability of tool cost	+1	+5	+3	+2	+6	+4
Standardization	+1	+1	+3	+2	+5	+4
Total cost	+12	+14	+21	+ 11	+25	+17

Cost of implementation system: The computation of this type of cost is demonstrated in Table 17 (lower part). The results show low costs for ATL and QVT-O, medium costs for ETL and Kermeta, and high costs for TGG and QVT-R.

Cost of poor reliability: The cost of poor reliability is computed based on six criteria including readability, writability, reliability, availability of tools, and standardization. QVT-O and ATL have better results. Kermeta and QVT-R have a high cost. Table 18 (upper part) presents the obtained results.

Table 18 The results of evaluating subject MTLs on cost of poor reliability (upper) and maintaining programs (lower)

Cost of poor reliability	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
Readability cost	+3	+5	+4	+1	+6	+2
Writability cost	+3	+1	+6	+5	+2	+4
Reliability cost	+4	+2	+5	+1	+6	+3
Availability of tool cost	+1	+5	+3	+2	+6	+4
Standardization	+1	+1	+3	+2	+5	+4
Total cost	+12	+14	+21	+11	+25	+17
Cost of maintaining programs	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
Readability cost	+3	+5	+4	+1	+6	+2
Writability cost	+3	+1	+6	+5	+2	+4
Reliability cost	+4	+2	+5	+1	+6	+3
Maintainability cost	+5	+1	+3	+2	+6	+4
Reusability cost	+6	+2	+4	+3	+5	+1
Availability of tool cost	+1	+5	+3	+2	+6	+4
Standardization	+1	+1	+3	+2	+5	+4
Total cost	+23	+17	+28	+16	+36	+22

Cost of maintaining programs: The cost of maintaining programs is related to several features of a language including readability, writability, reliability, maintainability, reusability, availability of tools, and standardization. The results are summarized in Table 18 (lower part). Due to these results, QVT-O and subsequently ETL have lowest costs in maintaining programs. After that, TGG and then ATL are more preferable than Kermeta and QVT-R.

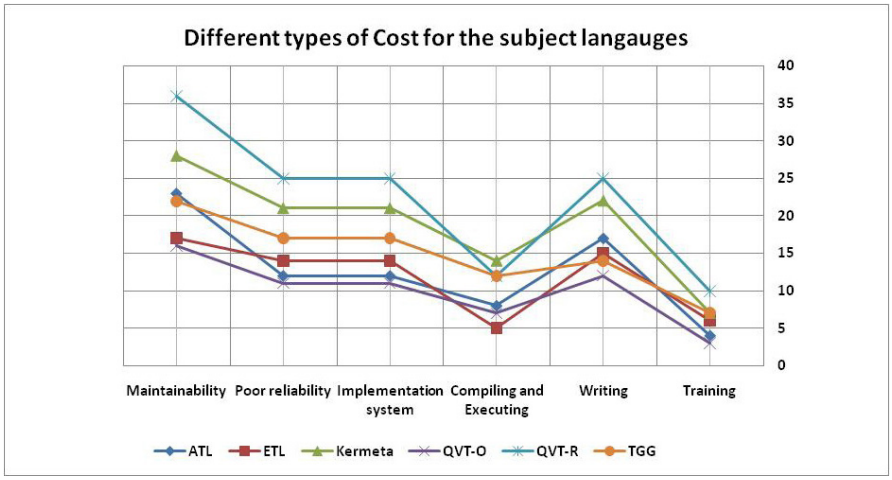

Figure 3. Comparison of the summarized results for different types of costs and for six subject MTLs.

In order to gain further insight to the results, the total amount for each of six types of costs and for each of six subject MTLs have been shown in Fig. 3. In this figure, the horizontal axis depicts six types of costs and the vertical axis depicts the amount of cost. Each line inside the figure belongs to an MTL. As can be seen, QVT-R has the highest cost for all types of costs except for the cost of compiling and executing. Moreover, QVT-O has the lowest cost in all aspects except for the cost of compiling and executing. Additionally, ATL and ETL have the lowest average costs with respect to different types of costs. Hence, from the cost point of view, ATL and ETL can be the best choices.

4.13 Comprehensive view of the results

In order to gain a comprehensive insight of all the subject MTLs with respect to all mentioned criteria except for cost, a radar diagram has been drawn which is shown in Fig. 4. The diagram helps a user for quick decision making about an appropriate language with respect to his/her requirements.

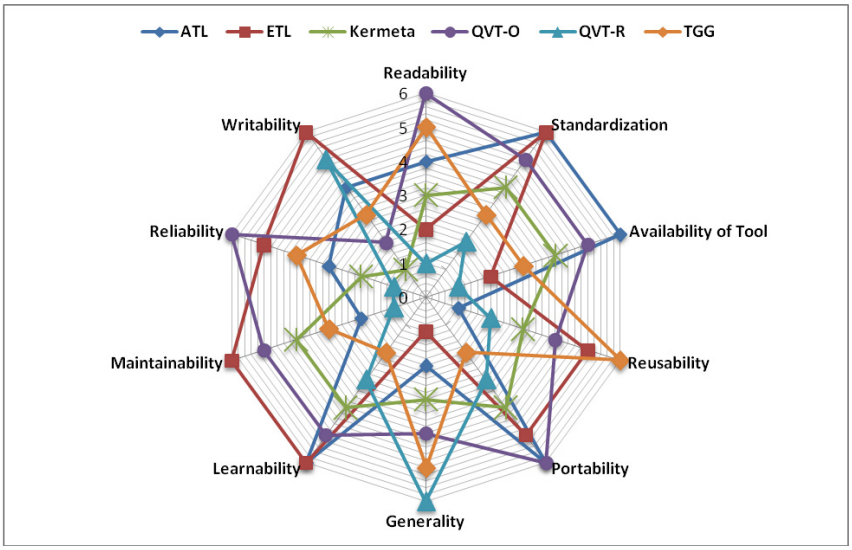

Figure 4. Comparison of six subject MTLs with respect to 10 major citeria in a radar diagram.

Each line belonging to a single MTL consists of 10 line segments and has rotated once inside the decagon. The line concerning each criterion starts from the center and ends at an external edge of decagon. The start point shows minimum value and the edge point shows the maximum value for each criterion. As an example, if a user needs a language with high writability, reliability, and learnability, ETL can be a good choice. This is because the line belonging to ETL is near or on the external edges of decagon for the three mentioned criteria. Otherwise, if availability of tools and standardization are more important for the user, the diagram suggests ATL. In case the user needs a language that partly satisfies all of the criteria, QVT-O would be a suitable choice due to larger area inside the decagon. As compared to QVT-O, Kermeta has smaller area yielding an average place from the evaluation criteria point of view.

Based on common sense, the presented 11 criteria are not all isolated; some of them actually overlap with others and some of them have conflict with others. Conflicting criteria[32] is of paramount importance when evaluating transformation languages. Each pair of conflicting criteria shows those criteria that cannot be optimized at the same time. According to the application at hand, one should establish an appropriate trade-off among each pair. In order to demonstrate the relationships between each pair of the criteria, Table 19 is prepared to show a brief picture of their correlation, whether they overlap or conflict each other. This table comprises 10 criteria and the cost is not incorporated.

Table 19 The correlation between each pair of criteria; '↑' strong overlap, '↑' weak overlap, '↓' strong conflict, '↓' weak conflict

	Readability	Writability	Reliability	Maintainability	Learnability	Generality	Portability	Reusability	Availability of Tools	Standardization
Readability	↑	↓↑	↑	↑	↓	↑		↓		
Writability		↑	↑	↑	↓	↑		↑		
Reliability			↑	↑	↓	↑		↑		
Maintainability				↑				↑		
Learnability					↑				↑	
Generality						↑				
Portability							↑			
Reusability								↑		
Availability of Tools									↑	
Standardization										↑

The observations from the table are as follows:

Readability. With writability, it has conflicts with rule scheduling and some of sub-criteria associated with syntax design; in other cases they are common. With reliability, they strongly overlap because reliability directly relates to readability. With maintainability, the same situation as reliability holds. With learnability, they weakly conflict because the number of primitives and the number of keywords improve readability and at the same time increase the size of the MTL. With generality, they are weakly overlapped because tracing sub-criterion in readability and traceability sub-criterion in generality are roughly sharing much commonality. With reusability, they weakly conflict because operator overloading is preferred in reusability while it degrades readability.

Writability. With reliability, they highly overlap due to direct relationship with writability. With maintainability, they have strong overlap due to commonality in the abstraction sub-criterion, which is of utmost importance. With learnability, they have weak conflict because the number of primitives and the number of keywords improve writability while at the same time increase the size of the MTL. With generality, they are strongly overlapped due to the commonality with tracing and bidirectionality

sub-criteria. With reusability, they have strong overlap because they are sharing abstraction, which is an important sub-criterion.

Reliability. With maintainability, they are strongly overlapped. The reason is that they are both sharing readability, which is considered immensely important. Moreover, maintainability of abstraction shares much commonality with writability, the sub-criterion of reliability. With learnability, they have weak conflict due to the dependence on readability and writability. With generality, they are weakly overlapped. With reusability, they have weak overlap because they are indirectly sharing abstraction.

Maintainability. With reusability, they are strongly overlapping due to much commonality in abstraction which is of utmost importance for both criteria.

Learnability. With availability, they have strong overlapping because they are sharing maturity which is considered important for both criteria.

4.14 Evaluation of the proposed methodology

In this section, we apply the methodology presented in Section 3.2 on six subject MTLs evaluated in the course of the Section 4.2 to 4.12. The results of the mentioned sections are used to derive the evaluation of the proposed methodology. Suppose that, there exist three kinds of *imaginary* users (stakeholders) in a software development team including *programmer*, *manager*, and *analyst*. Each of these users needs an MTL, but the requirements and preferred characteristics of each user concerning desired MTL differs. The users altogether nominate six MTLs for their tasks. However, they should agree on a single language. The candidate languages are those that were used as subject MTLs, i.e., ATL, ETL, Kermeta, TGG, QVT-O, and QVT-R. Using the proposed methodology is to determine the language with highest score for each user. Then, an MTL that has the highest score among majority of users is suggested as the most appropriate one.

The *imaginary* users are requested to specify the characteristics of their *preferred* MTL in terms of weighting the evaluation criteria and their sub-criteria. We offer five degrees of importance for each criterion or sub-criterion that are inoperative, weak, typical, high, and strong from the least important to the most important. These levels of importance are then converted to numerical weights between 0 and 2 such that 0 goes to inoperative, 0.5 to weak, 1 to typical, 1.5 to high and 2 goes to strong. Besides, the qualitative values of evaluating each criterion are converted to numerical values between 0 and 1 such that smaller numbers belong to less desirable values and larger numbers belong to more desirable values. Moreover, the range of 0 to 1 is divided by the number of values of each criterion. For example, a three-valued criterion is converted to 0, 0.5, and 1 accordingly. For criteria with high number of values, we need discretization and labeling beforehand.

Table 20 shows the numerical values for readability criterion and its sub-criteria and for six candidate MTLs. In order to save space, the values of other criteria are not presented. In addition, Table 21 though Table 23 show the *hypothetical* numerical weights concerning major criteria and their sub-criteria which are assigned by the *imaginary* users.

Table 20 The numerical values for the readability criterion and for six subject MTLs

Readability	ATL	ETL	Kermeta	QVT-O	QVT-R	TGG
Simplicity	0.25	0.50	0.72	0.22	0.47	0.58
Orthogonality	0.50	0.00	0.50	0.50	0.50	0.50
Syntax design	0.46	0.36	0.48	0.67	0.04	0.00
Syntactic separation	0.50	0.50	0.00	0.00	0.50	1.00
Application condition	1.00	1.00	0.00	0.50	0.50	0.50
Rule scheduling	0.50	0.50	1.00	1.00	0.50	1.00
Domain body	0.67	0.67	0.75	0.75	0.67	0.75
Tracing	1.00	1.00	0.00	1.00	1.00	1.00

Table 21 The *hypothetical* numerical weights concerning readability and writability, maintainability and reliability, learnability and portability, and standardization and reusability assigned by three *imaginary* users according to their requirements and preferences; for each criterion, Programmer (left), Manager (middle), Analyst (right)

Readability	**1.5**	**0.0**	**2.0**	**Writability**	**2.0**	**0.0**	**0.5**
Simplicity	2.0	0.0	2.0	Simplicity	2.0	0.0	2.0
Orthogonality	1.5	0.0	1.5	Orthogonality	1.5	0.0	1.0
Syntax design	1.0	0.0	1.5	Syntax design	1.0	0.0	1.0
Syntactic separation	2.0	0.0	1.5	Syntactic separation	2.0	0.0	1.0
Application condition	1.0	0.0	1.5	Application condition	1.0	0.0	1.0
Rule scheduling	1.0	0.0	1.5	Rule scheduling	1.0	0.0	0.5
Domain body	1.0	0.0	1.5	Domain body	1.0	0.0	1.0
Tracing	1.0	0.0	2.0	Tracing	1.5	0.0	2.0
				Abstraction	1.5	0.0	2.0
				Expressiveness	2.0	0.0	2.0
Maintainability	**0.0**	**1.0**	**2.0**	**Reliability**	**0.5**	**1.0**	**2.0**
Readability	0.0	1.0	2.0	Readability	1.5	1.0	2.0
Abstraction	0.0	1.0	2.0	Writability	2.0	1.0	0.5
Size	0.0	1.0	2.0	Rule strategy	1.0	1.0	1.5
Complexity	0.0	1.0	2.0	Exception handling	2.0	1.0	2.0
				Aliasing	1.0	1.0	2.0
				Domain typing	2.0	1.0	2.0
Learnability	**2.0**	**0.0**	**1.0**	**Portability**	**1.5**	**2.0**	**0.0**
Learning curve	2.0	0.0	1.0	Hardware (32/64bit)	1.0	2.0	0.0
Language size	2.0	0.0	1.0	OS (Win/Linux)	1.0	2.0	0.0
Maturity	1.5	0.0	1.0	Version (H/J/I/K/L/M)	2.0	2.0	0.0
Standardization	**0.0**	**2.0**	**1.0**	**Reusability**	**1.0**	**1.5**	**0.0**
Timeliness	0.0	2.0	0.0	Abstraction	1.5	1.0	0.0
Conformance	0.0	2.0	1.0	Rule inheritance	1.0	1.0	0.0
Obsolescence	0.0	2.0	0.0	Higher-order rules	1.0	1.0	0.0
				Composition	1.0	1.0	0.0

The results of scoring the six subject MTLs according to the specific requirements of three *imaginary* users have been shown in Table 23 and visualized in Fig. 5. As

can be seen, ATL best matches to the requirements of programmer due to its highest score. For manager, QVT-O achieves the highest score. For analyst, ATL is the best fit MTL. It turns out that ATL is an MTL which matches to the requirements and preferences of majority users. Therefore, it is suggested as an appropriate MTL to these three users.

Table 22 The *hypothetical* numerical weights concerning generality (left part) and availability of tools (right part) assigned by three *imaginary* users; In each part, Programmer (left), Manager (middle), Analyst (right)

Generality	**0.5**	**1.5**	**0.0**	**Availability of tools**	**2.0**	**1.0**	**1.0**
Source cardinality	1.0	1.0	0.0	Translator	2.0	1.0	1.0
Target cardinality	1.0	1.0	0.0	Editor	2.0	1.0	1.0
Technical space (EMF/XMI/XML)	1.5	1.5	0.0	Debugger	1.0	1.0	2.0
Bidirectionality	1.0	1.0	0.0	Easy installation	2.0	1.0	1.0
Traceability	1.0	1.0	0.0	History of use	2.0	1.0	0.0
Change propagation	1.5	1.0	0.0	Support for case studies	2.0	1.0	0.0
CRUD transformations	0.0	1.0	0.0	Technical support	2.0	1.0	1.0
Endogenous transformation	0.0	1.0	0.0				
Exogenous transformation	1.0	1.0	0.0				
Model merging	0.0	1.0	0.0				
Model comparing	0.0	1.0	0.0				
Model validating	1.0	1.0	0.0				
Integrated to code generator	1.5	1.0	0.0				

Table 23 The overall weights of the six subject MTLs for three *imaginary* supposed users

User	**ATL**	**ETL**	**Kermeta**	**QVT-O**	**QVT-R**	**TGG**
Programmer	0.69	0.66	0.58	0.64	0.47	0.58
Manager	0.54	0.47	0.49	0.57	0.41	0.50
Analyst	0.42	0.41	0.38	0.39	0.31	0.38

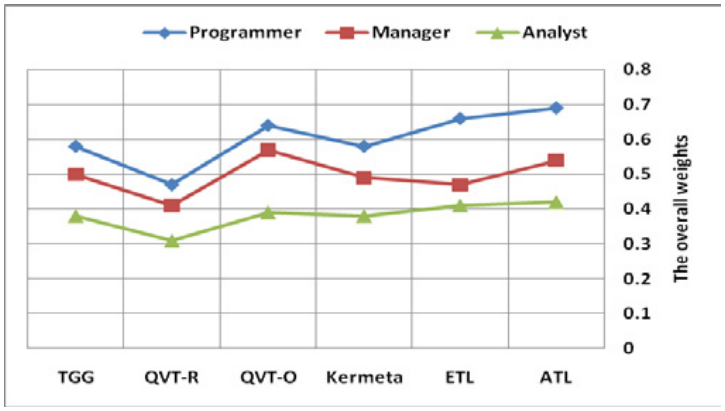

Figure 5. Comparison of six subject MTLs according to the specific requirements of three *imaginary* supposed users.

The reason why ATL achieved the highest score for programmer and analyst and QVT-O achieved the highest score for manager can be attributed to the considered weights or requirements of these users. As can be seen in Table 21, for manager, readability and writability are assigned the zero weight that in both cases ATL has higher scores as compared to QVT-O, while for programmer and analyst these weights are assigned a weight of two. Again, for learnability, the scores of QVT-O are less than ATL. However, due to the zero weight of learnability for manager, the difference of ATL and QVT-O in terms of learnability does not matter. The importance of generality criterion for manager coincides with the strength of QVT-O in this criterion which leads to increase the overall score of this language for manager. Note that for generality, QVT-R is even stronger than QVT-O. However, its weakness in terms of other criteria makes QVT-O superior to QVT-R for manager. For other criteria, the difference does not matter significantly.

Typically, after evaluation of MTLs using the mentioned criteria, it is expected that one can choose, among candidates, an appropriate MTL for the task at hand. However, this is not always trivial due to high number of criteria and features and diversity among them. For example, TGG and QVT-R have usually weak functionality in terms of the most criteria except for some limited ones. Therefore, the result of evaluations on these two languages can be guessed beforehand. However, this is not the case for stronger languages such as ATL, ETL, and QVT-O. This is why we need heuristics to assist in the process of evaluation. There might be cases that the results of evaluations using the MSR and interactive quantitative method (even for equal weights) do not equate. However, the benefit of the former is low complexity and faster evaluation, while the latter generates more accurate results. The accuracy of the results is advantageous especially in critical decision making situations where for example a manager must decide on the most appropriate language for the development team or a programmer that wants to start learning a new MTL.

It should come as little surprise that QVT-R gained the lowest score in our experiment for all users. In fact, this was expected due in large to lack of appropriate tools that can support all of the language's standards. Another reason lies in ambiguities in the language's standards.

In declarative languages such as QVT-R and TGG, the exact steps of transformations are not expressed and this makes ambiguities in the language especially for programmers that used to work with imperative languages such as Java. Interactive quantitative method enables a user to conduct some sort of what-if or goal seeking scenarios and study the effect of underlying parameters. The definition of weights in Eq. (1) and Eq. (2) for each of criteria and corresponding sub-criteria makes the interactive method of evaluation and decision making highly flexible.

As such, automating the interactive method could bring us a *decision-support* system in the context of MTL evaluation and selection. Even, one can add a machine learning method to the whole system for reasoning and forecasting on MTLS.

Overall, the experimental results of Section 4 suggest that accounting for insights achieved from evaluations using the introduced criteria, along with interactive quantitative method, has much potential when used as part of decision

making process for MTL selection. The results also illustrate that the choice of the suitable weights for criteria is strongly dependent upon the particular application of interest and can make many different results. For example, from an instructor's point of view who wants to teach an MTL to students, simplicity and learnability play essential roles. Traceability is also important for the teacher to enable him or her to make trace models and show what is transformed to what. Students, on the other hand, prefer debugging facilities of the language and tools. As another example, for a user who wants to implement a transformation in which model synchronization is required, the bidirectionality, incrementality, change propagation, and possibly model validation should be assigned a weight of 2 (strong level).

5 Discussion

Most of the model transformation and PL evaluation criteria, such as readability, writability, orthogonality, support for abstraction, type checking, portability, and reliability neither can be defined exactly, nor can be measured precisely. Nevertheless, they provide useful insights to the design and evaluation of these languages. This is why we had to apply a comparison method such as MSR strategy that enables us in part to analyze the relative superiority of a language to the others. Put simply: Evaluation of MTLs has been challenging, mainly because of: 1) difficulty in the measurement of the most criteria; 2) inaccurate results due to subjective measurements; 3) difficulty in combining the qualitative and quantitative measurements; 4) mutual effects and complex interrelationships among the existing criteria; 5) difference in the requirements of the users of a language; 6) difference in relative significance of the criteria in different situations.

When using the aforementioned criteria for evaluating PLs or MTLs, one should consider the inherent significance concerning each criterion. This is mainly because the significance (weight) of evaluation criteria is not the same; some criteria have high impact while the others may have low impact. For example, among readability, writability, and their sub-features, readability should be considered more important than writability. This is essentially due to the importance of the maintenance stage in the software development lifecycle. Since model transformations are also software developed using MTLs, the readability should be highly regarded when evaluating transformation languages. A transformation program is usually written once and might be read many times. Moreover, it may be partly modified to fix a problem or extend its functionality by a different developer. This needs reading the current program and understanding it. In addition to readability and writability, this point also holds for other paired criteria.

The importance of the evaluation criteria used for PLs and MTLs also varies according to the user of the language. For example, on the one hand, language implementers are mostly concerned with the details and issues of implementing the structures and features of a language. On the other hand, language users (programmers and developers) often think of writability and readability of languages. Language designers, as another user of a language, emphasize on the elegancy of the languages such that it gains wide-spread usage. These requirements and viewpoints often conflict with each other[32].

There exist some features such as testing and debugging facilities which are

provided by transformation language tools and IDEs. However, they influence the maintainability of the programs written with that language. In other words, testing and debugging are features that belong to the tools of a language, but they improve maintainability of the language itself.

In addition to the interactive approach mentioned in this paper, there can be an automatic approach in which we can specify a fixed and pre-defined set of stakeholders and their required criteria. Although this approach may provide the possibility of further investigation in advance, but in general, it reduces the flexibility of the overall approach.

Ease of verification of transformation programs written by an MTL is a desired attribute for a good language which leads in large to improve the reliability of transformation programs[42,61]. Ease of verification is a broad criterion which necessitates a separate investigation and is out of the scope of this paper. Statements ratio is a measure that can demonstrate the expressiveness of a language as compared to others[62]. Each line of code in high-level languages says more than that of low-level languages. This helps increase writability and productivity of the language. The bottom line is that establishment of the methods that are capable of precise measurement of criteria still remains an interesting research problem.

6 Related Work

Classification and Taxonomies of Transformation Approaches. Czarnecki and Helsen[33] proposed a framework for classification of model transformation approaches which is based on a feature model. Mens et al.[15] presented a multidimensional taxonomy of model transformations. It aims at helping a developer choose a specific transformation language, tool, method or technology according to his/her requirements based on the answers to some questions containing a set of concrete criteria. The results of the research by Taentzer et al.[31] generated a taxonomy and comparison of four MTLs that rely on graph techniques of transformations. Mohagheghi and Dehlen[63] provided a 7-step process to define an initial quality framework, adapted to model-driven engineering and applied to quality of model transformations. Mens et al.[64] suggested a taxonomy for graph transformations by formulating quality requirements. Various approaches to model transformation and desirable characteristics of an MTL have been presented by Sendall and Kozaczynski[65]. These studies present taxonomies, classifications, or quality frameworks for MTLs, often without case studies. Moreover, the MTLs used in their studies are either very limited or are not representative of large number of existing MTLs.

Evaluating Transformation Approaches. Huber[45] evaluated four MTLs, ATL, Kermeta, SmartQVT, and ModelMorf. The evaluation was accomplished via a taxonomy presented by Czarnecki et al.[33]. Lano et al.[5] provided a unified semantic treatment of model transformations, independent of any specific transformation language. The authors evaluated their framework by comparing the specifications of QVT-R, ATL, VIATRA, UML-RSDS, and Kermeta on three case studies including quality improvement, UML to relational database mapping, and tree to graph transformation. Some research has investigated the key factors that affect the quality of model transformations[7,66]. The purpose of these studies has

been to measure the quality of model transformation. Amstel et al.[7] defined a set of six quality attributes along with 27 metrics collected from six heterogeneous model transformations automatically. Vignaga[6] presented a set of 81 metrics to measure the transformations of ATL and assess their quality. The high number of metrics is due to the size and complexity of the ATL MM. The developed metrics by Vignaga do not apply to all transformation approaches and is almost dedicated to ATL. Measurements are objective and interpretations are subjective. By contrast, our method is general and can be applied on any sort of MTL including ATL. It has potential to compare different MTLs in a single category. PL criteria facilitate understanding the capabilities of an MTL even for novices in MDE context.

Rose et al.[67] conducted a comparison study on four model migration tools including AML[23], COPE, Ecore2Ecore, and Epsilon Flock. The authors utilized nine comparison criteria in a single dimension research. The chosen tools are specific for migration and the comparison criteria do not rely on a standard basis. A comparison of Epsilon Flock to other model migration languages has been performed by Rose et al.[68] in which they described Epsilon Flock, an M2M transformation language, suitable for model migration tasks. They used Ecore2Ecore, ATL, and COPE for their comparison with Epsilon Flock. Grønmo et al.[27] have compared three MTLs. The authors performed their comparison using a refactoring example which was eliminating unstructured cycles from UML activity diagrams. The aim of the research by Kapová et al.[69] has been to investigate the maintainability of transformations, specially written in QVT-R. They suggested 24 metrics for evaluations of model transformation generated in QVT-R and applied them to three common, but different transformations. Amstel[3,4] proposed ways of analyzing model transformations which are considered as objective measurements, such as dependency analysis and MM coverage. They have used three MTLs, ATL, QVT-R, and QVT-O on two common examples.

Kusel et al. presented[35] a survey and comparison of reuse mechanisms for MTLs using a comparative framework. This framework consists of major dimensions of reuse mechanisms and main stages in the process of reuse (including abstraction, selection, specialization, and integration). This study focuses on different sides of a single aspect of MTLs. Although reuse is an important mechanism to software development, it is only part of the story. A number of useful criteria should be incorporated into the evaluation process. Our method is designed to accomplish this goal by employing more familiar criteria. In case where someone needs to evaluate MTLs focusing on a single criterion and its sub-criteria, he or she can favour our decision-making methodology in combination with his or her criteria.

Schubert evaluated four different MTLs[46] including ATL, ETL, QVT-operational mapping language (QVT OML), and Xtend based upon the properties described in ISO 9126 standard. Kolahdouz-Rahimi et al.[2] established a systematic evaluation framework for MTL comparison, based upon the ISO/IEC 9126-1 quality characteristics for software systems. In their case studies, five transformation approaches QVT-R, ATL, Kermeta, UML-RSDS, and GrGen.NET were evaluated on a complex model refactoring case study. Contrary to this paper, the study of Kolahdouz-Rahimi et al. did not investigate any graph-based MTL.

[23] `wiki.eclipse.org/AML`

Additionally, it is only a comparison framework. After evaluating of MTLs by any method, the developer is responsible to decide on choosing an appropriate MTL based on the evaluation results. However, our method acts as a (semi) automatic decision-support system to bridge this important gap in the context of MTL evaluations.

Further investigating all the mentioned studies shows that few of them do not have any case studies and most of them performed one or two case studies. In addition, about half of studies used objective measurements and the others performed either subjective measurements exclusively or a combination of subjective and objective measurements. The number of evaluated MTLs in these studies varies between one to four that make results less generalizable. Some of evaluation studies are limited in terms of generality; they are MTL-specific[6], feature-specific[69], or both.

7 Conclusion

High number of MTLs with different capabilities and characteristics motivated the research community to find effective methods of evaluating and comparing MTLs to assist designers in making an appropriate choice for their task. The current methods relied on some criteria usually with subjective measurements. This paper proposed to utilize the PL criteria for evaluation of transformation languages. Deep understanding of various aspects of PLs resulted from applying well-defined criteria in comprehensive investigations motivated us to adapt them for evaluation of MTLs. This paper realized this intuition in detail and presented a family of criteria that facilitate the precise evaluation of transformation languages. We selected six MTLs as the representatives of all existing MTLs to be evaluated using the mentioned criteria. The results show the potential benefit of applying PL criteria to the assessment of MTLs. The PL criteria are strongly familiar for most of the programmers with various expertise and experiences. This deep understanding of criteria and prominent background of them leads to more precise measurements and eliminates the errors resulted from subjective evaluations. In addition, the criteria act as a *decision-support* framework for the designers during the development of software systems.

References

[1] Brambilla M, Cabot J, Wimmer M. Model-driven software engineering in practice. Synthesis Lectures on Software Engineering, 2012, 1: 1–182.

[2] Kolahdouz-Rahimi S, Lano K, Pillay S, Troya J, Van Gorp P. Evaluation of model transformation approaches for model refactoring. Science of Computer Programming, 2014, 85: 5–40.

[3] Van Amstel MF, Van Den Brand MGJ. Model transformation analysis: Staying ahead of the maintenance nightmare. Theory and Practice of Model Transformations, Springer Berlin Heidelberg, 2011: 108–122.

[4] Van Amstel M, Bosems S, Kurtev I, Pires LF. Performance in model transformations: Experiments with ATL and QVT. Theory and Practice of Model Transformations, Springer Berlin Heidelberg, 2011: 198–212.

[5] Lano K, Kolahdouz-Rahimi S, Poernomo I. Comparative evaluation of model transformation specification approaches. International Journal of Software and Informatics, 2012, 6 (2): 233–269.

[6] Vignaga A. Metrics for measuring ATL model transformations [Technical Report]. MaTE, Department of Computer Science, Universidad de Chile. 2009.

[7] Van Amstel MF, Lange CFJ, van den Brand MGJ. Using metrics for assessing the quality of

ASF+SDF model transformations. Theory and Practice of Model Transformations, Springer Berlin Heidelberg, 2009: 239–248.

[8] Wimmer M, Kappel G, Kusel A, Retschitzegger W, Schönböck J, Schwinger W, Kolovos DS. Surveying rule inheritance in model-to-model transformation languages. Journal of Object Technology, 2012, 11(2): 1–46.

[9] Samimi-Dehkordi L, Khalilian A, Zamani, B. Programming language criteria for model transformation evaluation. 2014 4th International eConference on Computer and Knowledge Engineering (ICCKE). IEEE. 2014. 370–375.

[10] Jouault F, Kurtev I. Transforming models with ATL. Satellite Events at the MoDELS 2005 Conference. Springer Berlin Heidelberg. 2006. 128–138.

[11] Kolovos DS, Paige RF, AC Polack F. Eclipse development tools for epsilon. Eclipse Summit Europe, Eclipse Modeling Symposium. 2006, 20062. 200.

[12] Drey Z, Faucher C, Fleurey F, Mahé V, Vojtisek D. Kermeta language reference manual. University of Rennes, Triskell Team. 2009.

[13] OMG QVT. Meta Object Facility (MOF) 2.0 Query/View/Transformation 1.2 Beta Specification, 2015.

[14] Schürr A, Klar F. 15 years of triple graph grammars. Graph Transformations, Springer Berlin Heidelberg, 2008: 411–425.

[15] Mens T, Van Gorp P. A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science, 2006, 152: 125–142.

[16] Biehl M. Literature study on model transformations [Technical Report]. Royal Institute of Technology. ISRN/KTH/MMK. 2010.

[17] Kolahdouz-Rahimi S. A comparative study of model transformation approaches through a systematic procedural framework and goal question metrics paradigm [PhD. Thesis]. King's College London (University of London), 2013.

[18] Epsilon Transformation Language. Copyright ©2014 The Eclipse Foundation. `http://www.eclipse.org/epsilon/doc/etl/`. 2014.

[19] Jouault F, Allilaire F, Bézivin J, Kurtev I. ATL: A model transformation tool. Science of Computer Programming, 2008, 72(1): 31–39.

[20] Kolovos DS, Paige RF, AC Polack F. The Epsilon transformation language. Theory and Practice of Model Transformations, Springer Berlin Heidelberg, 2008: 46–60.

[21] Jézéquel JM, Barais O, Fleurey F. Model driven language engineering with kermeta. Generative and Transformational Techniques in Software Engineering III, Springer Berlin Heidelberg, 2011: 201–221.

[22] SmartQVT Development Team, SmartQVT - a QVT implementation. `http://smartqvt.elibel.tm.fr/`. 2008.

[23] Gronback RC. Eclipse modeling project: A domain-specific language (DSL) toolkit. Pearson Education, 2009.

[24] Kiegeland J, Eichler H. Enabling comprehensive tool support for QVT. Eclipse Summit Europe, 2007.

[25] Stevens, P. A simple game-theoretic approach to checkonly QVT relations. Software & Systems Modeling, 2013, 12(1): 175–199.

[26] Willink E, Hoyos H, Kolovos D. Yet another three QVT languages. Theory and Practice of Model Transformations, Springer Berlin Heidelberg, 2013: 58–59.

[27] Grønmo R, Møller-Pedersen B, Olsen GK. Comparison of three model transformation languages. Model Driven Architecture-Foundations and Applications, Springer Berlin Heidelberg: 2009, 2–17.

[28] Hildebrandt S, Lambers L, Giese H, Rieke J, Greenyer J, Schäfer W, Lauder M, Anjorin A, Schürr A. A survey of triple graph grammar tools. Electronic Communications of the EASST, 2013, 57.

[29] Anjorin A, Lauder M, Patzina S, Schürr A. eMoflon: Leveraging EMF and professional CASE tools. Informatik, 2011: 281.

[30] Leblebici E, Anjorin A, Schürr A. Developing eMoflon with eMoflon. Theory and Practice of Model Transformations, Springer International Publishing, 2014: 138–145.

[31] Taentzer G, Ehrig K, Guerra E, De Lara J, Lengyel L, Levendovszky T, Prange U, Varro D, Varro-Gyapay S. Model transformation by graph transformation: A comparative study. Proc. Workshop Model Transformation in Practice. Montego Bay, Jamaica. 2005.
[32] Sebesta RW. Concepts of Programming Languages. 10th Ed. Pearson, 2012.
[33] Czarnecki K, Helsen S. Feature-based survey of model transformation approaches. IBM Systems Journal, 2006, 45(3): 621–645.
[34] Watt DA. Programming language design concepts. John Wiley & Sons, 2004.
[35] Kusel A, Schönböck J, Wimmer M, Kappel G, Retschitzegger W, Schwinger W. Reuse in model-to-model transformation languages: Are we there yet? Software & Systems Modeling, 2013, 14(2): 537–572.
[36] Wimmer M, Kappel G, Kusel A, Retschitzegger W, Schoenboeck J, Schwinger W. Surviving the heterogeneity jungle with composite mapping operators. Theory and Practice of Model Transformations, Springer Berlin Heidelberg, 2010: 260–275.
[37] Scott ML. Programming language pragmatics. Morgan Kaufmann, 2009.
[38] Al-Sibahi AS. On the computational expressiveness of model transformation languages. ITU Technical Report Series, 2015.
[39] Pressman RS. Software Engineering: A Practitioner's Approach, 7th Ed. McGraw-Hill Education, 2009.
[40] Lehrig S. Assessing the quality of model-to-model transformations based on scenarios [MSc Thesis]. University of Paderborn, Zukunftsmeile 1, 2012.
[41] Rensink A. The edge of graph transformation—graphs for behavioural specification. Graph transformations and model-driven engineering, Springer Berlin Heidelberg, 2010: 6–32.
[42] Pratt TW, Zelkowitz MV. Programming Languages: Design and Implementation, 4th Ed. Pearson, 2000.
[43] Farooq MS, Afzal Khan S, Ahmad F, Islam S, Abid A. An evaluation framework and comparative analysis of the widely used first programming languages. PloS One, 2014, 9(2): 1–25.
[44] Greenyer J, Kindler E. Reconciling tggs with qvt. Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, 2007: 16–30.
[45] Huber P. The model transformation language jungle: an evaluation and extension of existing approaches [Master's Thesis]. Business Informatics Group, TU Wien, 2008.
[46] Schubert LA. An Evaluation of Model Transformation Languages for UML Quality Engineering [Master's Thesis]. Georg-August-Universität Göttingen, 2010.
[47] Cuadrado JS, Guerra E, De Lara J. Uncovering errors in ATL model transformations using static analysis and constraint solving. 2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE). IEEE. 2014. 34–44.
[48] Rentschler A. Model transformation languages with modular information hiding. KIT Scientific Publishing, 2015, 17.
[49] ATL Transformations. `https://www.eclipse.org/atl/atlTransformations/#Class2Relational`. 2015.
[50] AtlanMod. `http://www.emn.fr/z-info/atlanmod/index.php/Main_Page/`. 2015.
[51] EMorF, Example-Zoo. `http://emorf.org/zoo/zoo.html`. 2012.
[52] Bézivin J, Dupé G, Jouault F, Pitette G, Rougui JE. First experiments with the ATL model transformation language: Transforming XSLT into Xquery. 2nd OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture. 2003. 37.
[53] Tisi M, Martínez S, Jouault F, Cabot J. Lazy execution of model-to-model transformations. Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, 2011: 32–46.
[54] Del Fabro MD, Valduriez P. Semi-automatic model integration using matching transformations and weaving models. Proc. of the 2007 ACM Symposium on Applied Computing. ACM. 2007. 963–970.
[55] Gerpheide CM, Schiffelers RRH, Serebrenik A. A bottom-up quality model for QVTo. 2014 9th International Conference on the Quality of Information and Communications Technology (QUATIC). IEEE. 2014. 85–94.
[56] Boronat A. Exogenous model merging by means of model management operators. Electronic Communications of the EASST, 2007, 3.

[57] Königs A. Model transformation with triple graph grammars. Model Transformations in Practice Satellite Workshop of MODELS, 2005, 166.
[58] Ermel C, Hermann F, Gall J, Binanzer D. Visual modeling and analysis of EMF model transformations based on triple graph grammars. Electronic Communications of the EASST, 2012: 1–12.
[59] Schürr A. Specification of graph translators with triple graph grammars. Graph-Theoretic Concepts in Computer Science, Springer Berlin Heidelberg, 1995: 151–163.
[60] Stevens P. Bidirectional model transformations in QVT: semantic issues and open questions. Software & Systems Modeling, 2010, 9(1): 7–20.
[61] Calegari D, Szasz N. Verification of model transformations: A survey of the state-of-the-art. Electronic Notes in Theoretical Computer Science, 2013, 292: 5–25.
[62] McConnell S. Code complete. Pearson Education, 2004.
[63] Mohagheghi P, Dehlen V. Developing a quality framework for model-driven engineering. Models in Software Engineering, Springer Berlin Heidelberg, 2008: 275–286.
[64] Mens T, Van Gorp P, Varró D, Karsai G. Applying a model transformation taxonomy to graph transformation technology. Electronic Notes in Theoretical Computer Science, 2006, 152: 143–159.
[65] Sendall S, Kozaczynski W. Model transformation: The heart and soul of model-driven software development. IEEE Softw., 2003, 20(5): 42–45.
[66] Van Amstel MF, Lange CFJ, van den Brand MGJ. Metrics for analyzing the quality of model transformations. 12th ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering. 2008.
[67] Rose LM, Herrmannsdoerfer M, Williams JR, Kolovos S, Garcés K, Paige RF, Polack FAC. A comparison of model migration tools. Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, 2010: 61–75.
[68] Rose LM, Kolovos DS, Paige RF, Polack FAC. Model migration with Epsilon flock. Theory and Practice of Model Transformations, Springer Berlin Heidelberg, 2010: 184–198.
[69] Kapová L, Goldschmidt T, Becker S, Henss J. Evaluating maintainability with code metrics for model-to-model transformations. Research into Practice–Reality and Gaps, Springer Berlin Heidelberg, 2010: 151–166.
[70] Klassen L, Wagner R. EMorF-A tool for model transformations. Electronic Communications of the EASST, 2012, 54.
[71] da Silva AR. Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems & Structures, 2015, 43: 139–155.
[72] Cadavid JJ, Combemale B, Baudry B. An analysis of metamodeling practices for MOF and OCL. Computer Languages, Systems & Structures, 2015, 41: 42–65.
[73] ATL Transformations. `http://www.eclipse.org/atl/atlTransformations/`. 2015.