Classification of Model Transformation Tools: Pattern Matching Techniques

Cláudio Gomes, Bruno Barroca, and Vasco Amaral

CITI, Departamento de Informática,
Faculdade de Ciências e Tecnologias,
Universidade Nova de Lisboa,
Portugal

Abstract. While comparing different model transformation languages (MTLs), it is common to refer to their syntactic and semantic features and overlook their supporting tools' performance. Performance is one of the aspects that can hamper the application of MDD to industrial scenarios. An highly declarative MTL might simply not scale well when using large models due to its supporting implementation. In this paper, we focus on the several pattern matching techniques (including optimization techniques) employed in the most popular transformation tools, and discuss their effectiveness w.r.t. the expressive power of the languages used. Because pattern matching is the most costly operation in a transformation execution, we present a classification of the existing model transformation tools according to the pattern matching optimization techniques they implement. Our classification complements existing ones that are more focused at syntactic and semantic features of the languages supported by those tools.

Keywords: Model Transformations, Languages Design, Pattern Matching Techniques

1 Introduction

The immersion of computer technology in a wide range of domains leads to a situation where the users' needs become demanding and increasingly complex (the problem domain). Consequently, engineering successful software systems also becomes increasingly complex (solution domain). A promising "divide-and-conquer" idea to break down this increasing complexity, is to intensively use Models during all stages of software development.

In Model Driven Development (MDD), both the design and development of new software systems is done by having multiple levels of abstraction, where each level deals only with a particular aspect of the system (therefore decreasing its complexity), and assuring the consistency between them (e.g., translations, synchronizations, etc.). In practice, each level of abstraction can be formalized by means of a Domain Specific Modelling Language (DSML), and materialized by its respective supporting tools ,i.e., editors, simulators, interpreters, analysers and compilers [1–3].

In this context, Model Transformation Tools (MTTs) are specifically designed to transform models according to a transformation specification expressed in a Model Transformation Language (MTL) [4]. Model transformation specifications are expressed by means of a set of symbolic representations of the source languages syntactic structures (also known as patterns) that represent is to be transformed during its execution.

Meanwhile, there exist so many different MTLs, with so many different properties and features, that anyone using them, here denominated as Transformation Engineer, can have serious problems selecting which one is the most appropriate to be used in a particular model transformation task. Moreover, the level of abstraction used on these MTLs, in practice, impacts both the productivity, and the scalability. In the one hand, it is known that the high level of abstraction employed in declarative MTLs, implies that model transformations expressed on them are not only easier to read and maintain by the Transformation Engineers [5]. In the other hand, this high level of abstraction *still* imposes a considerable downside on the run-time performance of the execution of the model transformations expressed on these MTLs [6–12]. This can be explained by the fact that the operation of finding the specified pattern in an arbitrary input model of the source language (also called pattern matching) is equivalent to the problem of finding a graph isomorphism [13, 14], which is an NP-Complete problem.

Run-time performance of MTLs is one of the aspects that can hamper the application of MTLs and consequently MDD, to industrial scenarios. In the general practice of software engineering, Transformation Engineers that use highly declarative MTLs in order to express their model transformations, can be forced to repeat themselves using imperative low-level programming languages, just because, it is still a major challenge for a declarative MTL to reach the point where its productivity outweighs its performance problems, at least compared with as imperative approach. We believe that research will continue to improve the performance aspect to the point where scalability will no longer be an issue. It is of utmost importance to provide the Transformation Engineer with a classification of the existing MTLs along with their supporting Model Transformation Tools (MTTs) in what matters to optimization techniques supported.

In this paper, we observe several Model Transformation Tools (MTTs), with particular focus on the implementation and optimization of the pattern matching techniques they employ.

This article contributes with the extensive collection of many different techniques ranging from the amount and kind of performance-related information required from the user, to the optimization techniques used on those tools.

In the next section, we present our methodology to select and classify existing MTTs. Then, in Section 3, we present some of the most used techniques, how they are organized according to our classification and which tools implement them. In Section 4, we explore our classification, how it can be used, and how it complements existing other existing classifications. Finally, we conclude in

Section 5 by relating the degree of optimization of each MTT with the syntactic features of the supported Model Transformation Language (MTL).

2 Methodology

In order to properly classify and compare the existing tools and their pattern matching techniques, we need to establish a common view and understanding of what is a model transformation environment and the execution process as a whole.

2.1 Transformation Environment Overview

A model transformation is "the automatic generation of a target model from a source model, according to a transformation definition" [15]. Fig. 1 establishes a common view of a generic MTL, its supporting tools and the involved models (input and output). Notice that all represented models are conforming to their respective metamodels.

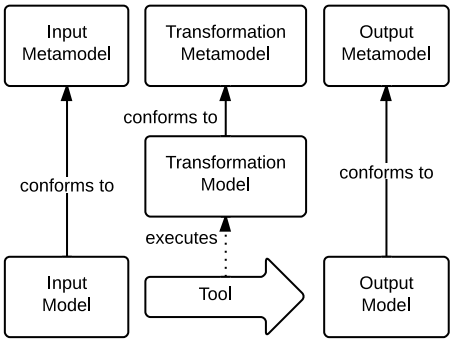
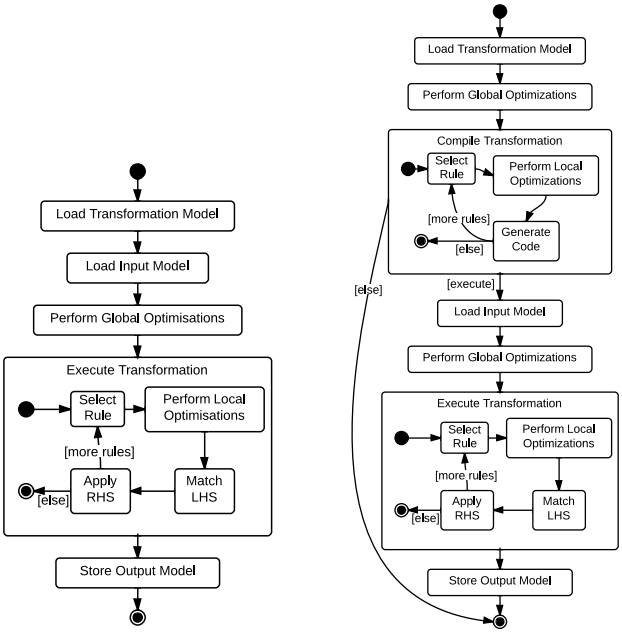


Fig. 1. Model transformation overview: language, tool and models.

2.2 Transformation Execution Overview

In order to improve pattern matching, existing tools employ optimizations at multiple stages of the transformation process. This is very similar to what happens in database systems where, prior to any query execution, there is index creation so that, when a query is made, other techniques such as exploring different evaluation plans, are applied to get the most efficient execution of that query [16]. It is because of that, that in order to study each pattern matching technique, we need an high-level description of a typical transformation execution highlighting the multiple stages of the process.



(a) Interpretation process of a transformation model.

(b) Compilation process of a transformation model.

Fig. 2. Execution process of a transformation model.

Fig. 2 identifies the two most followed approaches to MTL execution: interpretation (left) and compilation (right). We stress the fact that the presented diagrams are not supposed to describe exactly how model transformation tools operate, but instead to provide a clear view of the stages where optimization techniques can be employed. But these diagrams are general enough to fit even imperative tools (e.g., ATC [17]), where most of the stages are manually coded by the Transformation Engineer. We also assume that a transformation is comprised of a set of *rules*, each containing a Left Hand Side (LHS) pattern, that needs to be found in the input model, and a Right Hand Side (RHS) pattern which represents the model that will be output in the end of the execution. There is no loss of generality, since these *rules* (with the mentioned patterns) do not need to be explicitly represented in the MTL, but instead be explicit.

These Figures also show that an MTT always starts by loading the transformation and, in the particular case of interpretation, the input model. At this point, some existing MTTs perform **global** optimizations, such as the definition of indexes, refactorings, based on the analysis of the transformation specification. For instance, a **global** optimization may influence the order of rule selection, and even the information shared between different rules. Then the tool executes the transformation, selecting each rule and optionally performing some **local** optimizations. These optimizations are concerned with minimizing the search space

while searching for the occurrences of the LHS pattern in a given input model, i.e., while executing one rule.

We define **global** and **local** optimizations in terms of the scope of their impact. While **local** optimizations are concerned with improving one particular pattern match operation, **global** optimizations can impact several ones. Furthermore, **global** optimizations handle information from the whole input model, or from a representative one while **local** ones rely on limited information, either about the pattern in execution, or from some kind of aggregation provided by some other **global** technique.

The main difference between a compilation and an interpretation, from the point of view of the pattern matching process, is when the information input model is available to the tool. The two approaches have advantages and disadvantages: in interpretation mode, an MTT has to spend some cycles gathering information about the input model before executing the actual transformation process; whereas while compiling a transformation, a MTT can only access to statistics about typical input models. However, in order to compensate for the lack of information available during the compilation process, some tools can still prepare the generated transformation code so that the actual input model can be analysed when it gets executed. This means that, in both execution modes, the information about the input models can always be retrieved so, in principle, each optimization technique can be applied regardless of whether we are talking about a compilation approach or an interpretation approach. Of course there are techniques that do not depend on the information about the input model.

In summary, the possible optimization techniques that can be employed do not depend on the execution mode of the transformation tools, and so we do not need to classify the optimization techniques according to the execution approach in which they are employed.

2.3 Classification Rationale

In order to identify most of the existing pattern matching techniques, we tried to cover as many and as diverse MTTs as possible. We achieved variety on the different observed MTTs, by taking into account their distinguishing syntactic and semantic features as identified in [18], namely: (i) imperative tools such as ATC [17] and T-Core [19]; (ii) declarative tools such as AGG [20], Atom3 [21]; (iii) programmed graph rewriting approaches such as GReAT [22], GrGen.NET [23], PROGReS [24], VMTS [25] and MoTif [26]; (iv) incremental approaches such as Beanbag [27], Viatra2 [9] and Tefkat [28]; (v) and bidirectional approaches such as BOTL [29]. Notice that there are many more MTTs but we had to restrict our search to MTTs that published at least one paper about its internal execution mechanisms and optimization techniques. For instance, we did not consider tools such as SmartQVT [30] because we did not find any paper about optimization techniques being used in SmartQVT.

We followed a systematic approach to classify the pattern matching techniques. We first paid attention to the degree of domain and tool knowledge the Transformation Engineer has to have in order to perform (and/or improve) the

pattern matching execution of a model transformation. For instance, there are MTTs that require the Transformation Engineer to both manually code and improve the pattern matching procedure (these are called **manual approaches**). However, most of the existing MTTs do not require (or even allow) any intervention from the Transformation Engineer in the pattern matching procedure (**automatic approaches**). Yet, in an attempt to obtain the best of both worlds, there are MTTs whose languages introduce special syntactic constructs, so that if performance is at stake, the Transformation Engineer is able to interfere and optimize their execution (**semi-automatic approaches**).

We then organised the pattern matching techniques with respect to the scope of their impact. i.e., whether they are **global** or **local** optimizations.

We observed that some **local** techniques had a planning phase, were a cost model is used to perform the optimization, and some techniques rely on heuristics and hence, do not have a planning phase. These were classified in **planned** and **unplanned**.

These categories are just the foundation to classify most of the existing pattern matching optimization techniques. We found more categories while exploring a specific set of techniques. For instance, most **global** techniques are either **caching**, **indexing** or **overlapped pattern matching** techniques. In the next section we explore each of these categories and the concrete techniques with examples and referring to the state of the art MTTs that implement them.

3 Classification

Table 1 shows many MTTs and which pattern matching techniques they make use of. Since tools evolve very rapidly we have included the year next to the tool in which a paper was published concerning the tool's internal mechanisms to perform pattern matching. The header of Table presents the techniques we studied, organized according to the categories we introduced in the previous section. Notice that we do not include all the pattern matching techniques that we have found: we give special emphasis to those that are more pervasive across multiple MTTs.

We try to provide a simple and general explanation for each pattern matching technique with the help of the sample patterns shown in Fig. 4. We present each pattern as being matched against the input model shown in Fig. 3(a). Notice that, in the input model, the *id* attribute of each element appears to the left of its type.

3.1 Manual Techniques

MTTs that enable the Transformation Engineers to **manually** code and optimize the pattern matching process, usually do so by providing an API with the necessary imperative constructs to perform CRUD (Create-Read-Update-Delete) operations in the input model of the transformation. For instance, if the Transformation Engineer wants to match the pattern shown in Fig. 4(a)

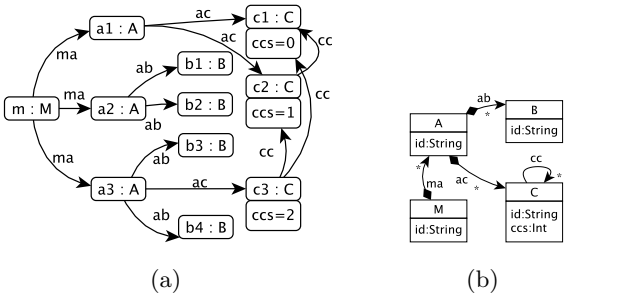


Fig. 3. Sample input model (left) and corresponding metamodel (right).

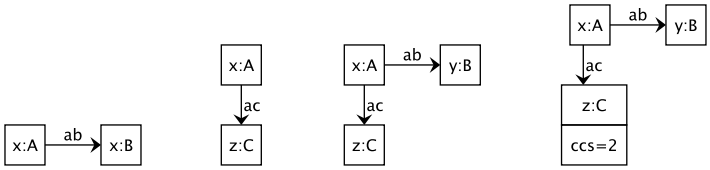


Fig. 4. Sample patterns.

against the input model of Fig. 3(a), s/he could leverage his/her domain knowledge by manually coding the pattern matching process as shown in Algorithm 1. The Transformation Engineer combines two crucial bits of information: *(i) domain knowledge*, since s/he knows that any instance of B is always contained in an instance of A (as it is shown in the metamodel of Fig. 3(b)); and *(ii) tool knowledge*, since s/he knows that the underlying model storage framework keeps inverse associations (such as *y.abInverse* in Algorithm 1). Also note that the *GetAllInstances(B)* operation is part of the tool's API, and is used to fetch the set of all instances of B in the input model of Fig. 3(a).

ATC [17] and T-Core [19] are good examples of low-level, imperative languages that require the Transformation Engineer to manually code the pattern matching process.

Due to the imperative nature of these languages, there are no optimization techniques that fall under the **Manual** category: if we want optimization, we have to do it ourselves.

3.2 Semi-Automatic Techniques

There is a wide array of **Semi-automatic** techniques such as the usage of *lazy rules* (as in ATL [31]), or the *user-specified strategies* employed to solve systems of equations involving several attributes (as in BOTL [32]). Most of the

Algorithm 1 A manually coded algorithm to match the pattern shown in Fig. 4(a) that takes advantage of the metamodel topology and the existence of inverse associations.

```
1: function MATCH
2:     for y ∈ GetAllInstances(B) do
3:         x ← y.abInverse
4:         results ← results ∪ {(x,y)}
5:     end for
6:     return results
7: end function
```

considered **Semi-automatic** techniques are exclusive of each individual tool but there is one that is pervasive in almost all the studied tools: **Pivoting**. It basically consists of reusing previously matched objects.

In order to support **Pivoting**, the MTL must include the necessary syntactic and semantic features that enable: *(i)* rule parametrization; and *(ii)* a way to instantiate those parameters with concrete model elements, effectively providing a starting point for the pattern matching process. It falls under the **Semi-automatic** category because the Transformation Engineer must identify which rules are suited to be parametrized, and forward previously matched model elements into those rules. As an example, let us assume that the pattern shown in Fig. 4(b) is matched before the pattern of Fig. 4(c). A keen Transformation Engineer parametrizes the pattern of Fig. 4(c) with elements that are to be matched in the pattern Fig. 4(b). Algorithm 2 shows the resulting generated code that matched the pattern of Fig. 4(c). For the sake of brevity, we omit the code generated to match the pattern of Fig. 4(b) as it would be similar to Algorithm 1. We also omit the generated code that calls the function defined in Algorithm 2 with the set of bindings collected during the matching process of pattern of Fig. 4(b).

GReAT [22, 11] and MoTif [33] allow for semi-automatic **Pivoting**. In those MTTs, a transformation specification consists of a network of rules with a well defined interface of input and output parameters. The input interface declares the rule's incoming partial matches that serve as a starting point for the pattern matching. The output interface represents the bindings that will be propagated to the next rules in the network.

3.3 Automatic Techniques

Most of the identified techniques are automatic, i.e., they require minimal intervention and knowledge from the Transformation Engineer in order to be used. The MTTs that employ these techniques typically work with declarative rules and, optionally, provide imperative constructs in order to enable the control (or configuration) of the rule scheduling. In what matters to the pattern matching process, we distinguish **Local** techniques from **Global** techniques.

Algorithm 2 An algorithm to match the pattern of Fig. 4(c).

```
1: function GeneratedMatch(AacCOccurrences)
2:     for (x,z) ∈ AacCOccurrences do
3:         for y ∈ x.ab do
4:             results ← results ∪ {(x,y,z)}
5:         end for
6:     end for
7:     return results
8: end function
```

The **Local** pattern matching techniques are algorithms that have to *transverse* the input model (in the worst case), while checking if there is a match *for each* element in the pattern.

Since their execution involves the selection of multiple choice points (i.e., possible candidate nodes to be checked), and going back to those choice points to test further alternatives, these algorithms are said to be local search based [34]. Even MTTs that reduce the pattern matching problem to a constraint satisfaction problem (e.g., AGG [12]), or even a database query problem (e.g., GrGen(PSQL) [35]) are indirectly performing local search [36, 16]. Because of this, they all fall in the category of **Local** optimization techniques.

While studying these techniques, we found that some involved a planning phase in which they use special data structures (such as search graphs in Viatra2 [9], or pattern graph in PROGRES [37]); and some simply execute the search immediately using nothing but some global data structures (such as indexes, as is done in VMTS [25]). We classified the former kind of techniques as **Planned** and the later as **Unplanned**. The referred data structures in **Planned** techniques are typically built automatically from LHS patterns, and provide a representation of all possible ways of searching for a given LHS pattern. For instance, a possible way to represent the search graph for the pattern of Fig. 4(d) is represented in Fig. 5(a). Notice that from the starting point, represented as a smaller circle, the local search can begin at either A, or B, or C. Suppose it starts from C, as indicated by the bold arrows, it then can proceed to A elements (maybe taking advantage of inverse relations), and so on for B elements. The bold arrows represent one of the many search plans.

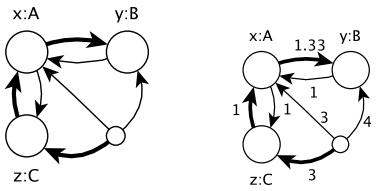
In order to be able to compare between search plans, and select a good one, MTTs use a cost model. We distinguish different **Planned** techniques according to the cost model used. Some cost models only use information about the metamodel (these are called **Metamodel Sensitive**); other cost models use statistical information about the input model (these are called **Model Sensitive**); or even explicit information about the tool's implementation (these are called **Implementation Sensitive**).

Metamodel Sensitive cost models employ a set of heuristics that make use of the match metamodel in a given model transformation—these were presented in [37], and used in the PROGRES tool. An example of such heuristics is the *first-fail* principle: a good plan should start the search in the most restricted

pattern element, since it will have the fewest possible occurrences. Following this principle, a good plan to search for occurrences of the pattern shown in Fig. 4(d) would be to start by searching all the z elements, given that they specify attribute constraints.

Model Sensitive cost models use statistical information about the current input model or, at least, of a representative collection of input models. As an example, we demonstrate the cost model used in the Viatra2 [9]. According to [9], the cost of a search plan is given by the potential size of the search tree formed by its execution. For instance, if we consider the search graph presented in Fig. 5(a), then the potential size of the search tree corresponding to a possible search plan shown in bold is given by $\bar{z} + \bar{z} * \bar{x_z} + \bar{z} * \bar{x_z} * \bar{y_x}$, where: $\bar{z}$ denotes the expected number of model elements that can be matched by the z element; $\bar{x_z}$ denotes the average number of model elements that can be matched by x after binding z to some model element; and $\bar{y_x}$ denotes the same for y after binding x to some model element. In Fig. 5(b), we show a weighted version of the presented search graph, but now considering the statistics of a given input model. Following the presented cost model (i.e., computing the potential sizes of the different search trees), it is clear that, in this case, the evaluation of a search plan that follows the order $z \rightarrow x \rightarrow y$ (shown in bold) yields a cost of 9.99, is preferable to a search plan that follows the order $y \rightarrow x \rightarrow z$, since its evaluation yields a higher cost of 12.

An **Implementation Sensitive** cost model such as the one presented in [6] and implemented in the GrGen.NET [23] tool, takes into account not only the size of the search tree, but also the cost of each individual operation such as the search for all the elements given some type. This allows the MTT to consider the existence of indexes and other characteristics of its own implementation in the cost model. This is similar to the cost model used in database systems since they typically take the indexes and other implementation features into account [16].



(a) (b)

Fig. 5. Search graph (left) and weighted search graph (right). These graphs represent the pattern of Fig. 4(d).

In what matters to **Global** techniques, we identify three different types of optimization techniques: **Caching**, **Indexing**, and **Overlapped Pattern Matching**.

Most of the analysed MTTs allow the definition of variables and conditions composed of multiple expressions over elements in a match pattern definition. Depending on the number of times that the same expression is used in different match patterns, its repeated evaluation may degrade the overall performance of the transformation. To mitigate this problem, transformation tools such as ATL [31], apply **Caching** techniques, by evaluating all expressions once, and storing the resulting values so that they can be directly retrieved later.

All of the observed MTTs use indexes. Most of the indexes used keep model elements grouped by their corresponding type as described in the metamodel (**Type Indexing**). However, we have identified two additional kinds of indexes: **Attribute** and **Structural**. While the former allows the MTTs to efficiently find elements given a condition on one of their attributes, the later allows MTTs to index whole patterns that are matched often in the transformation. In order to perform indexing, the required intervention and knowledge from the Transformation Engineers range from minimal (as in PROGRES [37, 24]), to none (as in Viatra2 [10, 9]), where in the later case, the **structural** indexes are automatically created for all of the patterns defined in a transformation specification.

Finally, there are MTTs that try to automatically factorize two or more match patterns, in order to identify a common pattern that can be matched before them. In this technique, known as **Overlapped Pattern Matching**, the common (or overlapped) pattern occurrences are then passed as pivots in order to be matched by the remaining patterns of the two rules [38]. Note that the difference between this technique and **Pivoting** is in the common occurrences detection, which has to be fully automated. If the user is required to identify common occurrences, then it is just **Pivoting**, as is done in Great [22, 11] and in Viatra2 [10, 9]. To the best of our knowledge, only VMTS [25] implements this technique.

Table 1. State of art tools and the pattern matching techniques in use.

	Manual	Semi-Automatic		Automatic									
		Local	Global	Local					Global				
			Pivoting	Planned				Unplanned	Caching	Indexing			Overlapped P.M.
				Model Sensitive		Metamodel Sensitive	Implementation Sensitive			Type	Attribute	Structural	
Tools				Multiple	Single								
PROGRES (1996)			X			X	X		X		X	X	
BOTL (2003)		X						X					
AGG (2004)			X		X	X				X			
Atom3 (2004)	X							X					
Great (2004)			X					X					
ATC (2006)	X												
Viatra2 (2006)	X		X	X	X	X				X		X	
Tefkat (2006)		X			X	X				X		X	
ATL (2008)	X	X						X	X				
GrGen.NET (2008)			X		X		X						
Motif (2008)			X					X		X			
BeanBag (2009)			X					X		X		X	
VMTS (2010)	X		X					X		X			X
T-Core	X												

4 Discussion

Performance is one of the aspects that can hamper the application of MDD to industrial scenarios. Before undergoing a major project, the Transformation Engineer should study which tools are better suited for that project. S/He can not risk choosing a declarative and productive MTL and, at a later stage of the project, discovering that the transformations specified in that language, supported by that tool, do not scale well.

Our classification in Table 1 complements existing ones by looking at the optimizations employed in the implementation of the MTLs. Of course this is a moving target: in theory, a language is independent of its implementation, so we expect that more and more optimizations will emerge that will outdate Table 1. However, we do not expect the kinds of optimization techniques as identified in Section 2, such as **Manual** vs **Semi-automatic** vs **Automatic**, **Local** vs **Global**, **Planned** vs **Unplanned**, and so on... to change that much.

Perhaps the most widely applicable categorization of MTLs is presented in [18]. Their MTL categorization is done in a comprehensive way by means of a feature model that elicits the variability of MTLs w.r.t. both their syntactic constructs and their semantic features. However, they do not make explicit any features regarding the run-time performance of their transformation engines.

On a more pragmatic point of view of MTL's usage context, [39] provides a taxonomy that aims to aid Transformation Engineers in deciding which MTL is best suited to carry out a particular model transformation activity. They identify as important characteristics the degree of automation and complexity. This taxonomy was extended in [40] by grouping several model transformation purposes (e.g., simulation, synchronization, optimization), according to the models, metamodels and abstraction levels involved in a given transformation.

In the context of quality engineering, [41] proposes a comprehensive evaluation schema based on ISO 9126 [42]. The proposed evaluation schema aims to help the Transformation Engineers on choosing an appropriate MTL, by comparing different MTL's tooling support, implementation, syntactic features, community support and their future perspectives. In a similar line of research, [43] compares four different MTLs and corresponding MTTs using a common transformation problem. Their categorization is based on the following characteristics: the representation of models and metamodels, the constructs used to define transformation rules, rule scheduling constructs and formal analysis support. They also take into account the tooling support for each language such as: Editors, Transformation Simulation Support, Compilation, Debugging and Validation.

To our knowledge, only [9] provides a categorization of graph transformation tools based in their pattern matching strategies but with focus in the execution of individual rules.

In summary, there is no classification directed at the underlying techniques employed in the transformation engines of the existing model transformation languages across the whole transformation process. Instead, most classifications are directed at their usability w.r.t their syntactic and semantic features and

usage scenarios. Knowing the optimization techniques supported by a given tool allows the Transformation Engineer to assess if that tool can be applied to industrial scenarios.

5 Conclusions

There is a wide variety of approaches to the pattern matching problem, having different outcomes, in what matters to the required amount of effort from the Transformation Engineer, and the end result in what matters to run-time performance. In this paper, we presented a classification of the different model transformation approaches w.r.t. the employed pattern matching techniques.

MTTs that focus on **Manual** approaches typically provide imperative (see Czarnecki's categorization [18]) constructs in their MTLs. Therefore, they require domain expertise and knowledge about the tool's internal pattern matching mechanisms. Their MTLs are powerful and expressive. However, the specified transformations are verbose and difficult to read, which hinders the productivity of the Transformation Engineer. Nevertheless, their execution can be extremely fast, since the Transformation Engineer is able to directly code any kind of optimizations using his/her knowledge about the domain. Therefore, these MTTs are ideally suited to perform critical model transformations, or even to implement higher-level MTLs, as is the case of T-Core [19], which was used to implement MoTif [44].

MTTs that support many **Automatic** pattern matching techniques require less amount of information from the Transformation Engineer and ease the creation and maintenance of the transformation specifications. The transformations are typically comprised of a set of rules declaring the manipulations in the input model without any information regarding the underlying required pattern matching process. Therefore, we can expect the maximum productivity of the Transformation Engineer. However, if the Transformation Engineer knows something about the domain that can be used to speed up the transformations' execution, he/she will not be able to use that knowledge because all the optimization decisions are made solely by the tool.

Finally, MTTs that support **Semi-automatic** techniques still require that the Transformation Engineer have some knowledge about their internal mechanisms, while enabling the expression of high-level declarative transformations. These tools typically focus on allowing the Transformation Engineer to modularize and parametrize rules (*Pivoting*) so that matched elements can be shared among them. The impact in the run-time performance is obvious: the initial bindings of a rule are automatically shared among the shared rules, decreasing (sometimes in several orders of magnitude) the amount of computation needed to match the remaining elements of a given pattern. However, in order to use this feature, the Transformation Engineer has to explicitly create the transformation with these features in mind in order to maximize the sharing of initial bindings between rules—i.e., this imposes a negative impact in the end productivity of the Transformation Engineer.

Ideally, one could expect that the problem of run-time performance of MTTs can be solved solely by **Automatic** pattern matching techniques. However, we observed that the approaches which invested more research in addressing the pattern matching problem tend to combine **Semi-automatic** and **Automatic** techniques. In the one hand, they allow the expression of model transformations as networks of rules, where the flow between rules represents shared LHS pattern occurrences; and in the other hand, they employ planned search in order to perform the pattern matching of the declared rules, while taking advantage of the shared LHS pattern occurrences.

We believe that further improvements on the state of the art regarding MTLs and their MTTs' run-time performance, will follow this research trend of powerful constructs to enable the tuning of model transformation specifications, and the configuration of the increasingly sophisticated pattern matching techniques.

References

1. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM systems journal **45**(3) (2006) 451–461
2. Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundation. Software, IEEE **20**(5) (2003) 36–41
3. Van Gorp, P.: Model-driven development of model transformations. Graph Transformations (2008) 517–519
4. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. Software, IEEE **20**(5) (2003) 42–45
5. Barroca, B., Amaral, V.: Asserting the correctness of translations. Electronic Communications of the EASST **50** (October 2011) ISSN=1863-2122.
6. Veit Batz, G., Kroll, M., Geiß, R.: A first experimental evaluation of search plan driven graph pattern matching. Applications of Graph Transformations with Industrial Relevance (2008) 471–486
7. Larrosa, J., Valiente, G.: Constraint satisfaction algorithms for graph pattern matching. Mathematical structures in computer science **12**(4) (2002) 403–422
8. Kalnins, a., Barzdins, J., Celms, E.: Efficiency Problems in MOLA Implementation. 19th International Conference, OOPSLA. Citeseer (2004)
9. Varró, G., Friedl, K., Varro, D., Schurr, A.: Advanced Techniques for the Implementation of Model Transformation Systems. PhD thesis, PhD thesis, Budapest University of Technology and Economics (2008)
10. Bergmann, G., Ökrös, A., Ráth, I., Varro, D., Varró, G.: Incremental pattern matching in the viatra model transformation system. Proceedings of the third international workshop on Graph and model transformations. ACM (2008)
11. Vizhanyo, A., Agrawal, A., Shi, F.: Towards generation of efficient transformations. (2004) 298–316
12. Rudolf, M.: Utilizing constraint satisfaction techniques for efficient graph pattern matching. Theory and Application of Graph Transformations (2000) 381–394
13. Mehlhorn, K.: Graph algorithms and NP-completeness. (1984)
14. Garey, M.R., Johnson, D.S.: Computers and Intractability. A Guide to the Theory of NP-Completeness. W H Freeman & Company (January 1979)
15. Kleppe, A.G., Warmer, J.B., Bast, W.: MDA explained: the model driven architecture: practice and promise. Addison-Wesley Professional (2003)
16. Silberschatz, A., Korth, H., Sudarshan, S.: Database System Concepts. McGraw-Hill Science/Engineering/Math (January 2010)
17. Estévez, A., Padrón, J., Sánchez, V., Roda, J.L.: ATC: A Low-Level Model Transformation Language. ... the 2nd International Workshop on Model ... (2006)
18. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. Volume 45 of Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture. (2003)
19. Syriani, E., Vangheluwe, H.: De-/re-constructing model transformation languages. Electronic Communications of the EASST **29** (2010)
20. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. Applications of Graph Transformations with Industrial Relevance (2004) 446–453
21. Lara, J.D., Vangheluwe, H., Alfonseca, M.: Meta-modelling and graph grammars for multi-paradigm modelling in AToM 3. Software & Systems Modeling **3**(3) (2004) 194–209

22. Balasubramanian, D., Narayanan, A., van Buskirk, C., Karsai, G.: The graph rewriting and transformation language: GReAT. Electronic Communications of the EASST **1** (2007)
23. Kroll, M., Geiß, R.: Developing Graph Transformations with Gr-Gen .NET. Applications of Graph Transformation with Industrial releVancE-AGTIVE **2007** (2007)
24. Schurr, A.: Progres, a visual language and environment for programming with graph rewriting systems. Technical report (1994)
25. Levendovszky, T., Lengyel, L., Mezei, G., Charaf, H.: A systematic approach to metamodeling environments and model transformation systems in VMTS. Electronic Notes in Theoretical Computer Science **127**(1) (2005) 65–75
26. Syriani, E., Vangheluwe, H.: A modular timed graph transformation language for simulation-based design. Software & Systems Modeling (2011) 1–28
27. Xiong, Y., Hu, Z., Zhao, H., Song, H., Takeichi, M., Mei, H.: Supporting automatic model inconsistency fixing. Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM (2009)
28. Lawley, M., Steel, J.: Practical declarative model transformation with Tefkat. Springer (2006)
29. Braun, P., Marschall, F.: Botl–the bidirectional object oriented transformation language. Technical report (2003)
30. Telecom, F.: Smartqvt: An open source model transformation tool implementing the mof 2.0 qvt-operational language, 2007
31. Jouault, F., Allilaire, F., Bezivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming **72**(1-2) (June 2008)
32. Braun, P., Marschall, F.: Transforming object oriented models with BOTL. Electronic Notes in Theoretical Computer Science **72**(3) (2003) 103–117
33. Syriani, E., Vangheluwe, H.: Programmed Graph Rewriting with DEVS. Applications of Graph Transformations with Industrial Relevance (October 2008)
34. Knuth, D.E.: The Art of Computer Programming, Volume 4A. Combinatorial Algorithms, Part 1. Addison-Wesley (January 2011)
35. Hack, S.: Graphersetzung für Optimierungen in der Codeerzeugung. Master's thesis, Universität Karlsruhe (2003)
36. Russell, S., Norvig, P.: Artificial intelligence: a modern approach. 2003. (2003)
37. Zündorf, A.: Graph pattern matching in PROGRES. Technical report (1996)
38. Mészáros, T., Mezei, G., Levendovszky, T.: Manual and automated performance optimization of model transformation systems. International Journal on ... (2010)
39. Mészáros, T., Van Gorp, P.: A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science (ENTCS **152** (March 2006)
40. Syriani, E.: A multi-paradigm foundation for model transformation language engineering. PhD thesis, McGill University (2011)
41. Schubert, L.A.: An Evaluation of Model Transformation Languages for UML Quality Engineering. PhD thesis, Master's thesis, Georg-August-Universität Göttingen, 2010.[cited at p. 101] (2010)
42. ISO/IEC: ISO/IEC 9126. Software engineering – Product quality. ISO/IEC (2001)
43. Ehrig, K., Guerra, E., Lara, J.D., Lengyel, L., Prange, U., Taentzer, G., Varro, D., et al.: Model transformation by graph transformation: A comparative study. In: IN MTIP 2005, INTERNATIONAL WORKSHOP ON MODEL TRANSFORMATIONS IN PRACTICE (SATELLITE EVENT OF MODELS 2005), 2005. ROHIT GHEYI, TIAGO. (2006) 71–80
44. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with time for simulation-based design. Theory and Practice of Model Transformations (2008) 91–106