On the Computational Expressiveness of Model Transformation Languages

Ahmad Salim Al-Sibahi
On the Computational Expressiveness of Model Transformation Languages

Ahmad Salim Al-Sibahi

IT University of Copenhagen
Rued Langgaards Vej 7
2300, Copenhagen
asal@itu.dk

Abstract. Common folklore in the model transformation community dictates that most transformation languages are Turing-complete. It is however seldom that a proof or an explanation is provided on why such property holds; due to the widely different features and execution models in these language, it is not immediately obvious what their computational expressiveness is.

In this paper we present an analysis that clarifies the computational expressiveness of a large number of model transformation languages. The analysis confirms the folklore for all model transformation languages, except the bidirectional ones. While this excludes many verification techniques from being applied on the languages as a whole, it might be possible to verify subsets of these languages; this holds especially for those languages which provide optional termination checking or those that require the use of exotic constructs to achieve the stated level of expressiveness.

Keywords: Model transformations, Turing-completeness

1 Introduction

Model transformation[11; 35] is one of the most central techniques employed in the field of model-driven engineering. A model transformation language allows the programmer to translate elements from one or more source models to elements in one or more target models, given their formal descriptions or metamodels. A classical example of such model transformation is the conversion of object-oriented class models to relational database models[23].

These model transformation languages can be either rule-based, where the programmer specifies how a subgroup of elements in the source model maps to the desired target elements, or purely imperative, where the programmer explicitly iterates through the source model and construct the desired elements of the target model.

For many of these languages however, it is unclear whether their computational capabilities[48] matches those of general programming languages, since
repetition in these model transformation languages can be implicit or bounded. Knowing the computational expressiveness of a language is important from a verification point of view since there are a large variety of techniques not applicable on Turing-complete computation systems. This paper presents a systematic analysis of the computational expressiveness of a wide selection of model transformation languages, and tries to pinpoint the constructs that enables or limits the level of expression.

1.1 Related Work

To the authors knowledge there has been no overall study on computational expressiveness of model transformation languages, however there exist several studies that compare different specific aspects. Gomes, Barroca, and Amaral [17] compare a variety of languages on their pattern matching capabilities in order to clarify the performance implications of each system. The work has served as a source of inspiration for finding interesting model transformation languages to investigate. A case study by Patzina and Patzina [37] compares different language features of ATL and SDM, and presents a table with information on whether these languages support recursion, in-place transformation and other capabilities. While significantly less comprehensive than this study, it supports many of the points made about the expressiveness of some languages presented in this paper. Finally, smaller comparisons with tables are presented in the related work sections of many model transformation language implementation papers e.g., Varró and Balogh [53] and Syriani and Vanghehwe [45]. These however often focus on a smaller subset of languages, and are often biased towards highlighting the novelty of their own languages.

2 Model transformation languages

Model transformation languages can generally be classified into multiple paradigms according to the underlying techniques they use. High-level imperative languages such as Xtend [16] and Rascal [25] try to provide features suitable for model-transformation such as dispatch functions and visitors, in addition to the many general-purpose constructs such as assignment, branching, and for- and while-loops. More low-level imperative languages like ATC [14] instead focus on easing the implementation of execution engines for rules-based languages, and therefore contain various components that can express core rule-oriented patterns.

Declarative model transformation languages are largely rule-based, and can be categorised further based on whether they mainly use graph rewriting as a formalism or not. Graph-based languages represent the source and target models using graphs—optionally with attributes and/or types—and describe transformations using sets of conditional graph rewrite rules which specify how subgraphs in the source map to subgraphs in the target. In these systems, running a transformation is equivalent to applying the rewrite rules until the target
output is produced (or divergence). Some graph-based languages like PRO-GReS [44], GReAT [3], GrGen.NET [22], Motif [45], VIATRA2 [53], VMTS [33] and SDM [15], use an explicit control flow where the programmer must imperatively specify the order of which the rules apply and when to repeat rule applications; other languages like AGG [47] and Atom3 [30] instead try to apply rules as long as applicable and the order of application is often decided by a combination of user-defined prioritisation of rules called layering and internally defined matching strategies.

Model-oriented rule-based languages focus on providing a set of declarative rules to describe a transformation between source and target models, given their meta-models. The rules in these languages often allows the user to use rich predicate languages like OCL [55] to constraint the scope of application, can explicitly call and dependent on application of other rules (even recursively) and they may allow a part of the transformation to be handled imperatively for more precision. Examples of these types of languages include ATL [24], ETL [26], Tefkat [31] and QVT [18].

Finally, bidirectional-languages work similarly to model-oriented languages except that transformations must be bidirectional. This concretely means that there should be a way to convert back from the target model to the source model, which can be made by making rules invertible as in BOTL [8] or using an explicit propagation mechanism as in BeanBag [56].

3 Computational expressiveness

3.1 Definition of Turing-completeness

The Church-Turing thesis [10] states that any effectively computable mathematical function can be computed by a Turing machine [20]. A system is said to be Turing-complete if it has the same power of expression as a Turing machine, or rather that it can compute any function a Turing machine can compute. Known examples of Turing-complete systems include the $\lambda$-calculus [2] and term-rewriting systems [41].

While it can be hard to pinpoint what features are sufficient for a programming language to be Turing-complete, there are some minimal criteria that must be fulfilled. First of all, a programming language must not have any memory bounds, analogously to extensibility of a tape in Turing-machines, and the lack of limit for term size in $\lambda$-calculus; a consequence of which it follows that the language must also support unbounded repetition such as recursion or iteration. Secondly, the language must be able to make different decisions on different types of input similarly to how a Turing machine conditionally moves it head or how a term-rewriting system can express a set of rules that are only applicable under certain conditions. Finally, the language must be able to change its current state and continue computation with a new state, whether by writing on the tape as the Turing machines, by reduction as in the $\lambda$-calculus or by replacement as in term rewriting.
3.2 General results on graph rewriting

As mentioned in Section 2, graph rewriting is the base formalism for many declarative model transformation languages; even some model-oriented rule-based transformation languages use graph rewriting implicitly in their implementation. Thus in order to understand the computational expressiveness of such languages, it is important to understand the expressiveness of general graph rewriting systems and which criteria must be satisfied to decrease or increase the expressiveness of such system.

Existing research shows that graph rewriting systems can express Turing-complete computation. An initial hypothesis could be that since term rewriting systems are similar to graph rewriting systems, they would have the same level of expressiveness. However, Plump [38, 39] argues that since graph rewriting systems are richer—possibly having cycles and shared nodes—such proposition might not be true; this fact is demonstrated by the author by translating a non-terminating term rewriting systems to a terminating graph rewriting systems. Instead, Plump proved Turing-equivalence of graph rewriting systems by encoding the Post correspondence problem.

### Post correspondence problem

The Post correspondence problem (PCP) [40] is a decision problem that describes a system where given a finite sequence of pairs of words

\[ [(a_1, b_1), \ldots, (a_n, b_n)] \text{ where } n > 1 \]

one has to tell whether there exist a sequence of indices

\[ [i_1, \ldots, i_k] \text{ where } i_j \in [1, n] \land j \in [1, k] \]

the following equality between concatenation of words (strings) hold

\[ a_{i_1} \cdots a_{i_k} = b_{i_1} \cdots b_{i_k} \]

For example, there exists a solution for the sequence \([bb, b), (ab, ba), (b, bb)\] namely the indices \([1, 2, 2, 3]\), because \((bb)(ab)(ab)(b) = bbabbb = (b)(ba)(ba)(bb)\).

Encoding PCP in a computation system is a common way of showing Turing-equivalence, since finding a solution to an arbitrary instance of PCP is undecidable.

Generally, a base requirement for graph rewriting systems to be terminating is that all the rules contained must either delete nodes or have application conditions [5, 38]. Any other graph rewriting system is trivially non-terminating since the same rule can be reapplied ad infinitum.

A common assumption in the field is that many rule-based graph and model transformation systems are often Turing-complete and can be translated to underlying graph rewriting formalisms [28, 47]. It is important to note that for model transformation languages, they are only directly translatable to general graph rewriting systems if they allow intermediate transitions and unbounded iteration. If a system can only do bounded iteration or bounded recursion, it
is trivially non-Turing complete since termination is decidable; similarly, if the 
rewrite system cannot analyse produced output or create intermediate state then 
there are only a finite amount of possible functions that can be expressed and 
therefore such system can neither be Turing-complete.

4 Methods of analysis

4.1 Methodology

The key focus of the presented analysis is to identify the computational expres-
siveness of various model transformation languages. The formal way to achieve 
that is to investigate whether it is possible to encode PCP or a Turing-complete 
system in each of the languages. However, such analysis would be infeasible to 
do for the number of investigated language, since that would require manually 
setting up and installing each system, learning each language sufficiently to per-
form a complete analysis, and verifying that such analysis is semantically correct. 
Furthermore, the failure to show the necessary encoding might not prove that 
the language is not Turing-complete and a further analysis of the whole system 
is needed to show the converse.

Instead, this paper focuses on performing a literature-based analysis whereby 
published articles, language manuals and other related documentation are used 
to check whether the stated languages contain combinations of features which 
are known to make them Turing-complete. If a formal (or semi-formal) semantics 
of the language is provided then that is used as a primary source; in most cases 
however, a combination of sources is required to get a complete overview of 
the language capabilities. This form of analysis should be sufficiently accurate, 
feasible to perform, and should be better reflective of what most ordinary users 
would learn when using the analysed model transformation languages.

4.2 Collection and identification process

The actual collection process proceeded as following:

- If a project explicitly mentions that the language is Turing-complete—often 
  providing a proof—then the result was noted including the example con-
  structs used.
- Otherwise, it was investigated whether the language contained the necessary 
  language constructs required to perform Turing-complete computation.

The constructs required to perform Turing-complete computation differ ac-
cording to the paradigm of the containing language. For languages with imper-
ative features, analysing the computational expressiveness is simple. If a system 
supports creation and substitution of variably-sized data, conditionals and un-
bounded looping (such as while-loops) or recursion, such language is Turing-
complete. If looping or data size is bounded, or it is not possible to branch then 
the expressiveness is limited at design time and results in a sub-Turing-complete
computation model. Note that for visual languages, blocks with internal repetition and blocks applied on collections are seen as looping constructs, and control flow that is transferred back to a previous block is seen as recursion.

For rule-based transformation languages with implicit control flow, such analysis is more subtle. The analysis must consider whether there exists limitations when describing rules in such system as whether possibly non-terminating rules are rejected or whether iteration must happen over a bounded collection. Even if a language does not permit explicit recursion and implicit repetition it can still be Turing-complete, e.g., if it permits rules that can invoke helper functions on some substructure or can express sufficiently complex constraints.

5 Analysis of computational expressiveness

5.1 General overview

Table 1 presents a comparison of the languages mentioned in Section 2 with regards to their core paradigm(s) and computational expressiveness. The paradigm columns tells whether a language contains rule-based constructs, imperative constructs or both. The languages with rule-based constructs are further partitioned into whether or not they are based on graph rewriting and whether they support specification of bidirectional rules.

Furthermore, the languages are partitioned according to the way they handle repetition, by whether they use explicit loops like for and while, whether they support some form of recursion or whether the repetition happens implicitly where the engine repeats specified rules until finding a fixed-point. Finally, the table presents whether such language given its features is Turing-complete based on the concrete analysis presented in Section 5.2.

5.2 Concrete analysis

In this section we will present a concrete analysis of each language justifying why they are Turing-complete or not.

**PROGReS** The language supports specification of arbitrary graph rewriting rules, whose execution is controlled by an imperative language [37; 42; 44]. Since the imperative control language has an unbounded looping construct loop [49], it can simulate a general graph rewriting system and is thus Turing-complete.

**AGG** It is possible to specify arbitrary graph rewriting rules in AGG [43; 46], including transformation on attributes, and the rules are run by an interpreter-based engine which only terminates when no rule is further applicable or is explicitly stopped by the user. Interestingly, the language supports checking whether a given rule-set results in a terminating program\(^1\), but such check is

\(^1\) Note that the analysis is semi-decidable: the programs that are accepted are definitely terminating, but it may also reject terminating programs if it cannot prove that they satisfy the required constraints.
<table>
<thead>
<tr>
<th>Language</th>
<th>Paradigm</th>
<th>Repetition</th>
<th>Turing-complete</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Imperative</td>
<td>Rule-based</td>
<td>General Graph-oriented</td>
<td>Bidirectional</td>
</tr>
<tr>
<td>PROGReS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>AGG</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>GReAT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>GrGen.NET</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Motif</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Atom3</td>
<td>AO1</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Tefkat</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>QVT Relations</td>
<td>QR1</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>QVT Operational</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>BOTL</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>BeanBag</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>VIATRA2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>VMTS</td>
<td>VM1</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ATL</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>ETL</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>SDM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Xtend</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Rascal</td>
<td>✓</td>
<td>RS1</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>ATC</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

- ✓: Bounded number of steps
- ✗: Status generally unknown
- AO1: Optional termination analysis
- AO: Constraints are specified by python statements
- QR1: Can call QVT Operational mappings
- VM1: Transformation on attributes using XSLT
- VM2: Optional termination analysis
- AT1: Using global helpers or lazy rules
- ET1: Using Epsilon Object Language (EOL)
- SD1: By setting up the required control flow and path expressions
- RS1: Pattern matching and visitors can simulate rules

Table 1. Expressiveness of model-transformation languages
optional and not enforced. Therefore, the language is considered Turing-complete as it corresponds to a general graph rewriting system.

**GReAT** The language [1; 3] supports blocks that perform rewriting on packets containing models. These blocks can be connected recursively in such way that the intermediate output of one block can be fed back to the input of a previous block, and a rule sequence is only terminating when no further output is produced. Therefore, the language can simulate a general graph rewriting system by choosing a general graph structure to be the meta-model of packets, make blocks perform graph rewriting rules and then use the unbounded iteration for execution; thus, the language is Turing-complete.

**GrGen.NET** The language user manual [6] shows an actual implementation of a Turing-machine. The system works on graphs and supports recursive rules [21], unbounded looping and general imperative constructs; all these features further supports the claim of Turing-equivalence.

**Motif** The language [45] works on graphs and supports unbounded looping (FRule, SRule), backtracking and recursion (XRule); and therefore the languages can simulate a general graph rewriting system. Furthermore, it is explicitly mentioned that these constructs can be used to make non-terminating programs, which further strengthens the argument that the language is Turing-complete.

**Atom3** The framework works with meta-modelling [52], and it possible for the user to specify transformations using graph rewriting rules at any level. The system is Turing-complete since these specified rules are run by a Graph Rewriting Processor (GRP) [30; 34] iteratively until no rule is longer applicable, and therefore the system can express any general graph rewriting system. The framework must also necessarily be Turing-complete, since Motif which is Turing-complete is implemented as a specialised language in the framework. Finally, the framework can execute arbitrary Python statements on attributes and can therefore also express Turing-complete computation in that way.

**Tefkat** This rule-based language [31] is based primarily on the second revised submission report on QVT by DSTC, IBM and CBOP [13] (with a few improvements), which explicitly states that such language is Turing-complete. This is because rules can specify complex constraints on both the source and target models, which in combination with unbounded recursion in rules makes it possible to perform arbitrary calculation.

**QVT Relations** The language [18] is Turing-complete since it can simulate general graph rewriting systems by in-place transformations where relations are reapplied on the source model until all required constraints are satisfied. Note that QVT Relations can further call QVT operational code and arbitrary external code using the QVT Blackbox mechanism.
**QVT Operational** The operational part of QVT [18] is Turing-complete since it is possible to specify recursive rules [29], in combination with creation of intermediate data, branching and looping. This makes it similar to many general-purpose imperative programming languages.

**BOTL** Model transformations in BOTL [8] are bounded by the number of defined rules and only applied to a finite set of matches. Therefore, all transformations are known to be terminating and therefore the language is not Turing-complete.

**BeanBag** The language [56] supports synchronisation of two models by using expressions containing various equational constraints, including variable binding, bounded iteration (forall, exists) and recursion. While various repetition constructs can be used, they must satisfy a stability property which ensures there always exists a valid synchronisation for given models. The only way to make non-terminating programs in the language is by using counter-intuitive circular equational constraints, and it is stated that most program in practice always terminate. Since it is unclear whether it is possible to exploit the circular constraints in an intuitive fashion to perform Turing-complete computation; without further formal analysis it is unknown if the language is Turing-complete. The formal analysis is left as future work, but a good place to start would be using the semantics presented in the main paper [56].

**VIATRA2** The language [4; 53] is based on two formalisms: graph transformation (GT) which allows the user to specify graph rewriting rules and abstract state machine (ASM) which is a known Turing-complete formalism [7; 19] and can be used to control the flow of execution. Furthermore, ASM supports an unbounded iterate instruction which in combination with the general features of GT makes it able to simulate a general graph rewriting system. Finally, it should be noted that the language also supports general recursion in pattern matching and that can be a source of non-termination for ill-formed programs; however, it is unclear whether recursive patterns can do actual computation and as such Turing-equivalence is unknown for that part.

**VMTS** The language [33] supports graph rewriting using a combination of visual model processors (VMP) and the visual control flow language (VCFL) [32]. VCFL supports unbounded recursion which makes the language a general graph rewriting system and thus Turing-complete. Additionally, transformations on attributes are performed using XSLT which is known to be able to perform Turing-complete computation [36]. Similarly to AGG, the language does however support optional checking for termination of a transformation.

**ATL** Ordinary declarative ATL [23; 51] rules cannot be recursive and are applied only a finitely amount of time by the execution engine [54], and by themselves they can not perform Turing-complete computation. However, it is possible to declare lazy rules [50] in ATL which are explicitly and possibly recursively called,
which in combination with the complex constraints on source and target models makes the declarative part of ATL Turing-complete in the same way as Tefkat.

In addition it possible to call the imperative helpers from the declarative rules, which further enables the language to perform Turing-complete computation. This is because helpers can be recursive [37] and in combination with the other imperative features (assignment, conditionals) can simulate a general-purpose programming language. Note that neither the imperative looping construct for nor OCL 2.0 expressions [9] are enough to perform Turing-complete computation, since they can only iterate through a finite collection of elements.

ETL Like all other languages in the Epsilon framework, the rule-based ETL [26; 27] uses a common core language called the Epsilon Object Language (EOL) for model transformation. EOL is described as a combination of JavaScript and EOL and contains all the common imperative language constructs from general-purpose programming languages like loops (both for and while), branching and assignment; therefore, the language is Turing-complete.

SDM Story Driven Modelling (SDM) [15] uses graph transformation as an underlying mechanism, where story patterns are used for describing additions and deletions to models similarly to graph rewriting rules, and a control flow language with conditional transfer via path expression is used to determine execution of patterns. The control flow language supports unbounded recursion [12; 37], and is therefore capable of expressing Turing-complete computation.

Xtend The language [16] is Turing-complete since it is a dialect of Java, which is a general purpose programming language. Like Java, it supports all commonly expected constructs for branching, assignment, looping and recursion.

Rascal This is a Turing-complete general purpose programming language [25] with focus on meta-programming. In addition to all expected imperative language constructs, it supports features like generic visitors and pattern directed functions which can emulate the main ideas of rule-based model transformation languages; therefore, the language can also simulate a general graph rewriting system.

ATC As a low-level language [14] for implementing model transformation engines, ATC supports many necessary primitives for model querying, matching, manipulation and transformation. In addition it contains common imperative language constructs, including unbounded while loops and therefore the language is Turing-complete.

6 Conclusion

This paper presented an analysis on a wide variety of model transformation languages, confirming that most of these languages are Turing-complete. While such
property was not always obvious, it was confirmed to hold for all languages investigated except the bidirectional ones: BOTL and BeanBag. Since the primary features that enabled Turing-completeness for each language were identified, it might still be possible to use more advanced verification techniques on programs that do not use these features. Finally, languages like AGG and VMTS have optional termination checking that ensures sub-Turing-completeness of specific programs, and thus further eases the verification of these programs.

Acknowledgements

We would like to thank Aleksandar Dimovski for the initial inspiration for this report, further discussions and final review. We would further like to thank Andrzej Wasowski for helpful feedback during the writing process. This report is supported by The Danish Council for Independent Research under the Sapere Aude scheme, project VARIETE.

References


REFERENCES


REFERENCES


