Generating Model Transformation Rules from Examples using an Evolutionary Algorithm

Martin Faunes
DIRO, Université de Montréal
Montreal, Canada
faunescm@-
iro.umontreal.ca

Houari Sahraoui
DIRO, Université de Montréal
Montreal, Canada
sahraouh@-
iro.umontreal.ca

Mounir Boukadoum
Univ. de Québec à Montréal
Montreal, Canada
mounir.boukadoum@-
uqam.ca

ABSTRACT

We propose an evolutionary approach to automatically generate model transformation rules from a set of examples. To this end, genetic programming is adapted to the problem of model transformation in the presence of complex input/output relationships (*i.e.*, models conforming to metamodels) by generating declarative programs (*i.e.*, transformation rules in this case). Our approach does not rely on prior transformation traces for the model-example pairs, and directly generates executable, many-to-many rules with complex conditions. The applicability of the approach is illustrated with the well-known problem of transforming UML class diagrams into relational schemas, using examples collected from the literature.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Miscellaneous

General Terms

Algorithms

Keywords

Model transformation by example, genetic programming

1. INTRODUCTION

The success of Model Driven Engineering (MDE) depends heavily on automating model transformation (MT). MT allows transforming a source model that represents a system from some point of view, or at some level of abstraction, into a target model that asserts facts of the same system, but from another point of view or at another level of abstraction, using a transformation program typically consisting of rules [7]. The source and target models must conform to corresponding source and target metamodels which define the point of view and the level of abstraction to use when expressing models for specific systems.

In recent years, considerable advances have been made in modeling environments and tools. From the point of view of the transformation infrastructure, a large effort has been made to define languages for expressing transformation rules and thus make the writing of transformation programs easier. However, having a good transformation language is only one part of the solution; the most important part is to define/gather the knowledge on how to transform any model conforming to a particular metamodel into a model conforming to another metamodel. For many problems, this knowledge is incomplete or not available. This is the motivation behind the research on learning transformation rules.

The idea of Model Transformation by Example (MTBE) consists of deriving transformation programs by generalizing concrete transformations found in a set of prototypical examples of source and target models. Initially proposed by Varró in [8], this idea is gaining popularity with many recent contributions (*e.g.,* [1, 3, 4, 6, 9]). These approaches solve the problem of rule derivation only partially. Some of them require detailed mappings between the examples of source and target models, which are often difficult to provide. Others cannot derive rules that test more than one construct in the source model and/or produce more than one construct in the target model (many-to-many rules), a requirement in many transformation problems. A third limitation is the inability of some approaches to produce complex rule conditions to define precise patterns to search for in the source model. Finally, some approaches produce abstract, non-executable rules, which makes it difficult to validate them empirically.

This paper describes an approach for example-based rule generation. This apporach does not require detailed mappings between models and produces executable many-to-many rules with complex conditions, thus allowing coverage of a large spectrum of transformations between pairs of metamodels. To build this approach, genetic programming (GP) is adapted to the particular problem of MTBE. GP allows evolving programs in order to improve their ability to approximate a behavior that is defined by a set of valid pairs of inputs/outputs. In the context of MTBE, the programs are transformation rule sets, and the behavior is defined by example pairs consisting of source and corresponding target models. The approach was evaluated on the well-known transformation problem of UML class diagrams into relational schemas. The obtained quantitative and qualitative results show that almost all the involved constructs are correctly transformed.

2. APPROACH

Model transformation by example (MTBE) [8] can be summarized as follows: Considering a set of transformation pairs <source model, target model> given as examples, derive a set of transformation rules that would transform an arbitrary source model into its corresponding target model. For instance, given a set of transformations of UML class diagrams into corresponding relational schemas, derive a set of rules that would transform *any* diagram into a schema.

Regardless of its nature, a generic rule derivation process takes two inputs, the source and target metamodels, and the transformation example pairs. The rule derivation process could be simplified at the cost of providing extra inputs, *e.g.,* detailed mappings traces between source and target example models, or limiting the types of transformations that can be learned, *e.g.,* no many-to-many transformation rules. In the first case, one has to make assumptions about the availability of the extra inputs, which could be unrealistic in many cases. Similarly, when restricting the types of transformations that can be derived, the MTBE approach may become inapplicable to important pairs of metamodels.

Existing MTBE approaches do not produce complex transformations and/or require detailed mappings in the examples. This work proposes a novel MTBE solution that does so. The solution derives transformation rules without the need for transformation traces; it produces many-to-many rules with complex and various types of conditions; finally, the derivation process is guided by the ability of the generated rules to successfully transform the provided examples, which guarantees that they are executable with the right control sequence. The reminder of this section presents and discusses the proposed approach.

2.1 Overview

Because transformation rules are declarative programs, MTBE can be seen as an automated process to derive such programs. This observation motivates adapting genetic programming (GP) [5] to the MTBE problem. GP is a technique widely used to derive mostly imperative programs by specifying a program's behavior from a set of inputs and expected outputs examples. The initial motivation of GP is that it is easier to give examples of inputs/outputs than to manually write a program, which is the case of many transformation problems. The derivation process is done by iteratively improving an initial population of randomly-created programs, *i.e.*, by reproducing the fittest programs. The reproduction is made by means of *genetic operators* similar to the ones observed in nature.

Before, starting the GP cycle, the user must have a set of example pairs describing the expected program behavior in the form of <inputs, outputs>. The user must also define a way to encode and create the initial population of random programs. Finally, a mechanism is needed to run the programs on the provided inputs and compare the execution results with the expected outputs. This is typically done by defining a fitness function that evaluates a distance or error between the produced and expected outputs. The first step of a GP cycle consists of creating the initial population of programs. Then, the programs of the current population are run on the input examples to produce the expected outputs to evaluate their fitness. If the current population satisfies a stop criterion, (*e.g.*, a predefined number of iteration cycles or a target fitness value) the best obtained program is returned; otherwise, the fittest programs are selected for reproduction. Although, the selection process favors the programs with the highest fitness values, it includes a random part to allow for novelty. Reproduction involves three families of operations: (i) *elitism* to directly add top-ranked programs to the new population, (ii) *crossover* to create new programs by combining the genetic material from old ones, and (iii) *mutation* to alter an existing program by randomly adding new genetic material. Once a new population is created, it replaces the current one and the next iteration of the GP cycle takes place. Thus, the programs progressively changes to better implement the specified behavior.

2.2 Genetic Programming Adaptation

GP applies mostly to imperative programs, generally represented by trees, and takes input-output examples as simple structures. GP must be adapted to the MTBE problem since the goal is to produce declarative programs (rules) and input-output examples with complex structures (models conforming to metamodels) are used.

Typical transformation problems require a set of transformation rules to cover all the patterns in the source models. A program p is encoded accordingly as a set of transformation rules, $p = \{r_1, r_2, ..., r_n\}$. Each transformation rule r_i is, in turn, encoded as a pair $r_i = (SP, TP)$, where SP is the pattern to search for in source models and TP is the pattern to instantiate when producing the target models.

The source pattern SP is a triple, $SP = (GSC, SJC, G)$, where GSC is a set of generic source constructs, SJC is a set of join conditions, and G is a guard. A generic source construct is the specification of an instance of a construct type that has to be matched with concrete constructs in the source model. Each generic construct has the properties of its construct type in the source metamodel. When matched with a concrete construct from the source model, these properties take the values of the latter. Join conditions SJC allow to specify a source pattern as a model fragment, *i.e.*, a set of interrelated constructs according to the metamodel. The Guard G defines a complex condition on the properties of the generic source constructs. It is encoded as a binary tree containing elements from both terminal (T) and primitive (I) sets. T is the union of the properties of the constructs in GSC and a set of constants C that depends on the property definition domains. The set of primitives I is composed minimally of logical operators and comparators ($I = \{and, or, not, =, >, <, ...\}$). Other operators, such as arithmetic or string operators, could be added to test values derived from the basic properties.

The target pattern TP is a triple $TP = (GTC, B, TJC)$, where GTC, B and TJC represent respectively a set of generic target constructs, a set of binding statements, and a set of join statements. A generic target construct specifies a concrete construct to create in the target model when the rule is fired. The set of bindings B determines how to set the property values of the created constructs from the property values of the constructs that match the source pattern. Finally the join statements allow to connect the created constructs to form a fragment in the target model.

For the initial population, a number of rule sets NRS is created (NRS is a parameter of the approach). The number of rules to create for each rule set is selected randomly from a given interval. For each rule, we define a random procedure to create the source and target patterns. To cre-

ate a source pattern SP, a generic construct set GCS of a given size is selected from the source metamodel. This is accomplished by a random walk through the metamodel such that, each time a construct type is reached, a corresponding generic construct is added to GCS. The set of join conditions SJC is defined as the links between constructs in the random walk. To define a random guard G to complete the source pattern, a tree is created by randomly mixing elements from the terminal set T, *i.e.*, properties of the selected constructs and constants consistent with their types, and elements from the primitive set P of operators. The creation of the tree is done using a variation of the "grow" method defined in [5]. The creation process must ensure the type consistency between the operations and the operands.

The creation of a target pattern TP follows the same random walk principle as for the source pattern, but with the target metamodel as the walked through structure. This step allows creation of the generic target constructs and the join statements between them. The binding statements are generated by randomly assigning elements in the terminal set T to the properties of the generic target constructs that were not set by the join statements. The random property assignment is done according to the property types.

During the evolution process, all the generated rules are translated into JESS, a fact-based rule language, and executed using its rule engine [2]. As the target model is created by means of fact assertions, it is possible for the same construct to be created several times by different rules. For example, if a class $C1$ is at the origin/destination of many associations, each time a rule that transforms an association is fired, a table is created for $C1$. However, as the creations consist of asserting the same fact, the table is created only once. This particularity of JESS is an advantage in this work since it helps in rule execution control. There is no need to check if a construct already exists before executing a rule.

In GP, the fitness function usually measures the difference between the produced and the expected outputs from a program. Accordingly, the fitness function in this work measures the difference between the target models produced by a rule set and the expected ones as indicated in the example model pairs. The fitness of a rule set is defined as the average of the transformation correctness of the model-example pairs. This average could be weighted by the number of constructs in the target models. Comparing a produced model with the expected one to evaluate the transformation correctness (*i.e.*, comparing two graphs with typed nodes) is a difficult task. Considering that in the proposed GP-based rule derivation, the fitness function is evaluated for each rule set and population (iteration), and for each example pair, this cannot afford exhaustive graph comparisons. Instead, a quick and efficient graph kernel g is used to compare the target model M in the example pair and the target model M' produced by the rule set when transforming the source model of the same pair. g is defined as follows:

$$g(M, M') = \phi g_f(M, M') + \psi(1 - g_n(M, M')) \rightarrow [0, 1] \quad \phi, \psi \leq 1; \phi + \psi = 1 \tag{1}$$

g_f calculates the percentage of constructs in M and M' that exactly match. Two constructs exactly match if they are of the same type and have equal values for all their properties. g_n measures the percentage of constructs that were not exactly matched nor by-type matched. A construct in M (respectively in M') have a type match if there exists a construct not exactly matched with the same type in M' (respectively M). The percentages could be calculated regardless the construct types or as a weighted average by construct type. In this last case, construct types with few, or lot off, instances are equally considered.

In GP, a population of programs is improved by applying genetic operators (mutation and crossover). These operators are defined according to the problem to solve and should guarantee that the derived programs are syntactically and semantically valid. Before applying the genetic operators, the programs are selected depending on their fitness values. Some of them could be duplicated directly into the new population without reproduction, *i.e.* elitist strategy. In this work, *roulette-wheel* selection is used. This technique assigns to each rule set in the current population a probability of being selected for reproduction that is proportional to its fitness. This selection strategy favors the fittest rule sets while still giving a chance of being selected to the others.

The crossover operation consists of producing new rule sets by combining the existing genetic material. It is applied with high probability to each pair of selected rule sets. After selecting two parent rule sets for reproduction, two new rule sets are created by exchanging parts of the parents, *i.e.*, subsets of rules. For instance, consider the two rule sets $p_1 = \{r_{11}, r_{12}, r_{13}, r_{14}\}$ having four rules and $p_2 = \{r_{21}, r_{22}, r_{23}, r_{24}, r_{25}\}$ with five rules. If cut-points are randomly set to 2 for p_1 and 3 for p_2, the offspring obtained are rule set $o_1 = \{r_{11}, r_{12}, r_{24}, r_{25}\}$ and rule set $o_2 = \{r_{21}, r_{22}, r_{23}, r_{13}, r_{14}\}$. Because each rule is syntactically and semantically correct before the crossover, this correctness is not altered for the offspring.

After crossover, the obtained offspring could be mutated with a small probability. Mutation allows the introduction of new genetic material while the population evolves (by randomly altering existing rules or adding new ones). For the transformation problem, mutation could occur at the rule set level or at the single rule level. Each time, a rule set is randomly selected for mutation, a mutation strategy is also randomly picked. Two mutation strategies are defined at the rule-set level: *adding a randomly-created rule* and *deleting a randomly-selected rule*. At the rule level, many strategies are possible. For a randomly-selected rule, one could *add or delete a generic construct in the source pattern*, *change the condition tree*, *add or delete a generic construct in the target pattern*, and *change the binding* between source and target constructs. These strategies are implemented preserving the syntactic and semantic validity of the produced rules.

3. EVALUATION

To illustrate the ability of our approach to derive transformation rules from examples, let us consider the transformation of UML class diagrams into relational schemas. This transformation is not trivial and requires many-to-many rules with conditions on the properties values. The examples used in this evaluation were collected from published material. 12 examples were selected with the goal of covering different application domains in order to simulate real situations. The average size of the class diagrams was 30 constructs, the smallest diagram having 19 constructs and the largest 43.

To test the proposed approach, leave-one-out cross validation was used. The rule derivation process was executed 12 times using, each time, a different example for testing and

the 11 remaining ones as examples. For each execution, the class diagram of the testing example was transformed using the obtained rules, and the produced relational schema was manually checked with respect to the schema given in the example. The manual checking consisted of matching constructs of the produced schema with those of the expected one and vice versa. Then, each construct was classified as "fully matched"(FM), "partially matched"(PM), or "not matched"(NM).

Figure 1 show the results of the 12 leave-one-out runs. These results are good in terms of FM (72%) and average for NM (23%). Similar scores were obtained with and without considering construct types in the calculation of g. The percentage of matched constructed could be improved if we increase the size of the population and/or the number of generations. For instance, we obtained an FM of 75% by increasing the size of the initial population. For a sanity check, we compared these results with those obtained by randomly generating a large number of rule sets and by selecting the best set. The difference is statistically significant for both FM ($p < 0.001$) and NM ($p = 0.023$).

Upon examining the generated rules, we noticed that frequent situations are generally well captured. The generated rules correctly transformed classes into tables, attributes into columns, and unique attributes into primary keys. Transforming associations is often impaired by cardinality problems. For instance, in the case of many-to-many associations, a rule was found to create a table for the association which is the right transformation. However, in its guard, rather than testing that both cardinalities were greater than one, the rule tested the cardinalities for equality instead. This happened because the examples contained a large number of many-to-many associations and very few 1-to-1 associations. Consequently, the error of generating tables for one-to-one associations was not frequent enough to heavily penalize the fitness function. For 1-to-many associations, a rule was found that generates a foreign key with the wrong table references. The trend to map these associations to foreign keys was sound, but the property binding was not done properly. Similarly, for inheritance, we found rules that created the expected constructs but with many wrong property values. Finally, some rules generate the right constructs in the target model for the tested source pattern, but they generate unnecessary constructs in it as well.

In conclusion, the ability to derive correct rules for particular situations depends on the representativeness of these situations in the example base. Moreover, the well-formedness of the produced target models should be better enforced to avoid constructs with incorrect join properties such as columns without tables. Finally, a refinement step is needed to detect rule subsumption, redundancy, and inconsistencies.

4. CONCLUSION

We propose an approach for deriving model transformation rules from examples, based on Genetic Programming. Rule sets are first randomly generated and then evolved through crossover and mutation in order to better approximate the desired transformation. The evolution is guided by examples that give an indication on how the transformation should behave. The novelty of the approach is that it does not require detailed transformation traces, but only a set of example models pairs. Additionally, it does not restrict the types and complexity of the generated rule sets. Finally,

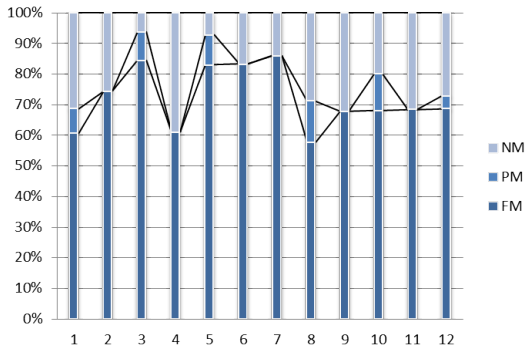

Figure 1: Quantitative Results with $c3$

possible transformation rule sets are executed as they are generated to ensure the efficiency of transformations.

The preliminary results obtained show that the rule derivation approach performs significantly better than random generation. They also show that for the particular case used in the evaluation, the quasi-majority of the transformation rules was retrieved using a relatively small set of examples.

Future research include evaluate the approach over other transformation problems. And also, to show its universality, it is intended to evaluate it on problems that require sophisticated rules such as non-trivial property value derivation.

5. REFERENCES

[1] X. Dolques, M. Huchard, C. Nebut, and P. Reitz. Learning transformation rules from transformation examples: An approach based on relational concept analysis. In *Int. Conf. on Enterprise Distributed Object Computing Workshops*, pages 27 –32, 2010.

[2] E. F. Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications, 2003.

[3] M. Kessentini, H. Sahraoui, and M. Boukadoum. Model transformation as an optimization problem. In *Int. Conf. on Model Driven Engineering Languages and Systems*, pages 159–173. 2008.

[4] M. Kessentini, M. Wimmer, H. Sahraoui, and M. Boukadoum. Generating transformation rules from examples for behavioral models. In *Proc. of the 2nd Int. Workshop on Behaviour Modelling: Foundation and Applications*, pages 2:1–2:7, 2010.

[5] J. Koza and R. Poli. Genetic programming. In *Search Methodologies*, pages 127–164. 2005.

[6] H. Saada, X. Dolques, M. Huchard, C. Nebut, and H. Sahraoui. Generation of operational transformation rules from examples of model transformations. In *Int. Conf. on Model Driven Engineering Languages and Systems*, 2012.

[7] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, 20(5):42 – 45, 2003.

[8] D. Varró. Model transformation by example. In *Int. Conf. on Model Driven Engineering Languages and Systems*, pages 410–424. 2006.

[9] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler. Towards model transformation generation by-example. In *Annual Hawaii Int. Conf. on System Sciences*, pages 285b–, 2007.