Extracting Correspondences from Metamodels Using Metamodel Matching

Shichao Fang[0000−0002−3556−4346] and Kevin Lano[0000−0002−9706−1410]

Department of Informatics, King's College London, London, UK
{shichao.fang, kevin.lano}@kcl.ac.uk

Abstract. In Model-Driven Engineering (MDE), metamodels define the structure of software models such as Petri Nets. This paper proposes a new approach to extract correspondences from metamodels, in order to automatically derive transformations on models. We present the approach on an example of two versions of metamodels for Petri Nets, and evaluate it on benchmark examples of metamodel matching.

Keywords: Model-Driven Engineering · Model Transformation · Metamodel Matching.

1 Introduction

In the Model-Driven Engineering (MDE) paradigm, software systems are specified and maintained as *models*, whose structure is defined by *metamodels*. Eg., Figure 1(a) is a metamodel of a simple Petri Net language, whose models consist of sets of linked places and transitions within nets. Model transformation (MT) is the most important process of MDE. Transformations are a set of rules defining how to transform a source model to a target model. One example of a MT language is QVT-R [10]. A correspondence of two metamodels defines a mapping between classes of the metamodels, and between properties (class features) in the source classes and target classes. These correspondences can be used to define transformations.

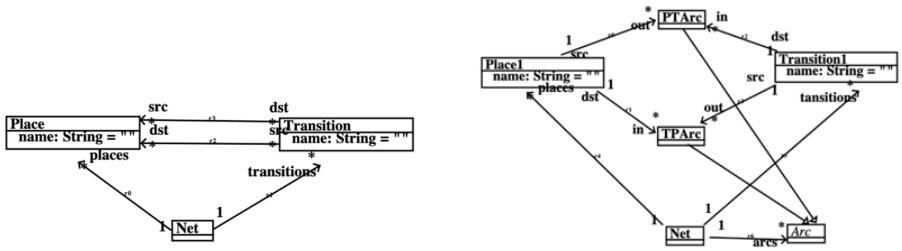
In this paper, we propose an approach to extract correspondences between a source metamodel and target metamodel. First, this requires transforming classes of both metamodels to a normalised form (which explicitly contain all their inherited and composed features). Secondly, we calculate the similarity of each pair of classes, and thirdly determine correspondences based on all similarities. We have evaluated its accuracy using several examples including the Petri Net case (Fig. 1(a) and Fig. 1(b)) of [11]. In addition, we have evaluated the quality of the approach by using the benchmark cases of [12].

2 Related Works

A few tools have been created for the synthesis of model transformations from metamodels. AML [3] requires a user to specify metamodelling matching strategies, but this needs high knowledge of metamodelling. Only ATL transformations

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2405/02_paper.pdf



(a) version 0 (b) version 3

Fig. 1. Petri Net metamodels

are produced. COPE [4] supports metamodel-specific migrations. However, for complex metamodel matching, they still need developers to manually match metamodels.

For similarity measures we considered several different alternatives: *Graph structural similarity (GSS)* tests if two classes have similar graph structure metrics [8] in the different metamodels. *Graph edit similarity (GES)* evaluates the graph edit distance of the reachability graphs of 2 classes in the 2 metamodels [2]. *Name syntactic similarity (NSS)* measures the string edit distances [7] of the names of two classes. *Name semantic similarity (NMS)* identifies if class names are synonymous terms or in the same/linked term families according to a thesaurus [6]. *Semantic context similarity (SCS)* applies measures of ontological similarity between the 2 metamodels [9].

Graph similarity measures treat a metamodel as a graph of nodes (classes) and edges (associations, aggregations and inheritances).

We evaluated the different measures on a large collection of different metamodel pairs. We found that data-structure similarity (DSS) was the consistently best approach. GSS overall has a poor average. GES is quite accurate, however it has exponential time complexity [2]. NSS can be misleading – eg., in the WebML and EER case of [5] *Relationship* in WebML corresponds to *RelationshipEnd* in EER, not to its namesake. Likewise, NMS is more useful for cases where there is a common vocabulary.

The state of the art in metamodel matching is represented by [5], who use evolutionary algorithms with NSS/NMS to search for possible class and feature matchings. However they do not consider DSS or composed features in their matchings, but only non-composed (directly owned) and inherited features. Transformation synthesis is not addressed in [5]. We consider DSS is a better basis for producing transformations. Using a deterministic procedure is preferable, since the results of evolutionary algorithms can vary from run to run. It was feasible to apply deterministic search to some examples of [5] with comparable results (Table 3).

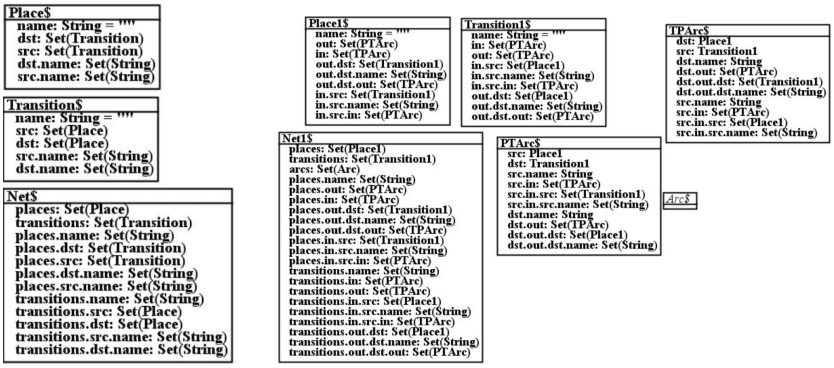
3 Flattening Classes

Flattening a class is the process of representing the class in a form which represents all of its recursively inherited and composed features [1]. For a class, in addition to its owned properties, its other properties are derived from inheritance and navigation. The process of flattening inheritance is: for each class C, if C has a superclass D, copy properties of D to C if they are not already in C. To flatten navigation, the process is: for each class C, if it has an association r to another class D, add properties $r.a$ (for each property $a : T$ of D) to C. Specifically, Table 1 summarizes the types of $r.a$ under different conditions.

Table 1. Types of $r.a$ for $a : T$

Type	Condition
T	if r has 1-multiplicity
Set(T)	if r has *-multiplicity and T is not a collection type
Sequence(T)	if r has * ordered-multiplicity and T is not a collection type
Set(S)	if r has *-multiplicity and T is a collection type *Set(S)* or *Sequence(S)*
Sequence(S)	if r has * ordered-multiplicity and T is *Sequence(S)*

These two steps are repeated until there is no change in the metamodel. The associations are also represented as properties in the flattened representation, which ensures that the structure of the metamodel is fully expressed in the flat version. The flattening process terminates when reaching loops in the graph. Fig. 2 shows the flattening results on the Petri Net metamodels. The flattened class of a class C is named C\$.



(a) source metamodel (b) target metamodel

Fig. 2. Flattened classes of Petri Net metamodels

4 Calculating Similarity

In assessing similarity, it is only necessary to look at the types of the properties, not at their names. This kind of similarity is called *data structure similarity*

(DSS). Although sometimes property names are very similar, names may be too much influenced by human factors. For example, two completely different properties in different metamodels can have the same name if the developer chooses. In contrast, the types of properties are much more objective.

To make the similarity result more accurate, we need to consider all possible matches of classes. Eg., if we assume that class *E1* matches class *E2*, then type *E1* is considered fully similar (value 1 equivalent) to type *E2*. However, equality of types can be fuzzy, for example, a type E property with 0..1 multiplicity could be considered 0.75 similar to a type E property with 1 multiplicity. This allows more flexible matches than strict equality of the types. In this paper we use fixed similarity values for types (see Table 2).

Table 2. Type similarity measures between different types of properties *a, b*

Similarity	*a : T*	*a : Set(T)*	*a : Sequence(T)*
b : T	1	0.5	0.5
b : Set(T)	0.5	1	0.8
b : Sequence(T)	0.5	0.8	1

For two flattened classes $A\$$ and $B\$$, the data structure similarity (DSS) is:

$$DSS = \frac{SumOfTypeSimilarities}{max(A\$.properties.size, B\$.properties.size)}$$

5 Metamodel Matching and Transformation Derivation

We sum the individual class DSS similarities to obtain a mapping score. However, sometimes there are multiple mappings with the same highest map score. In this case, calculating *name similarity* is used to choose one map. To calculate the name similarity, we use string edit distance [7]. This measures the minimum number of operations (insertions, deletions, substitutions) which change one string into another one. For two strings A and B, the name syntactic similarity (NSS) is:

$$NSS = \frac{A.size + B.size - ed(A, B)}{A.size + B.size}$$

The correspondences are expressed as mappings of classes and their features. Eg.: $Transition\$ \longmapsto Transition1\$$ with

$$name \longmapsto name \quad src \longmapsto in.src \quad dst \longmapsto out.dst$$

where $Place\$ \longmapsto Place1\$$. From these mappings, transformations in different MT languages can be synthesised. So far, we generate QVT-R, QVT-O and UML-RSDS. Eg., in QVT-R the *Transition* mapping is:

```
top relation MapTransition2Transition1
{ enforce domain sourc t : Transition
  { name = n, src = t_src : Place {}, dst = t_dst : Place {} };
  enforce domain targ t1 : Transition1
```

```
  { name = n, in = t1_in : PTArc { src = t1_in_src : Place1 {} },
    out = t1_out : TPArc { dst = t1_out_dst : Place1 {} } };
  when
  { Transition2Transition1(t,t1) and
    Place2Place1(t_src,t1_in_src) and
    Place2Place1(t_dst,t1_out_dst) }
}
```

6 Evaluation

In the Petri Net example, the source metamodel contains 3 classes, and the target metamodel contains 5 classes, meaning there are 60 potential class mappings. After calculating these 60 mappings, we found 2 mappings with the highest map score, 1.6825396825396826. The first mapping is: *Transition$* $\mapsto$ *Transition1$, Net$* $\mapsto$ *Net1$, Place$* $\mapsto$ *Place1$*. The second mapping is: *Transition$* $\mapsto$ *Place1$, Net$* $\mapsto$ *Net1$, Place$* $\mapsto$ *Transition1$*. This arises because of the structural symmetry of the *Place* and *Transition* classes – they can be validly interchanged according to DSS.

By calculating name similarity, the first result is selected as the best mapping, and its detailed correspondences are as follows:

- *Matching for Transition$ and Transition1$*: [name, src, dst, src.name, dst.name] $\longmapsto$ [name, in.src, out.dst, in.src.name, out.dst.name]. Similarity of *Transition$* and *Transition1$* is: 0.5555555555555556.
- *Matching for Net$ and Net1$*: [places, transitions, places.name, places.dst, places.src, places.dst.name, places.src.name, transitions.name, transitions.src, transitions.dst, transitions.src.name, transitions.dst.name] $\longmapsto$ [places, transitions, places.name, places.out.dst, places.in.src, places.out.dst.name, places.in.src.name, transitions.name, transitions.in.src, transitions.out.dst, transitions.in.src.name, transitions.out.dst.name]. Similarity of *Net$* and *Net1$* is: 0.5714285714285714.
- *Matching for Place$ and Place1$*: [name, dst, src, dst.name, src.name] $\longmapsto$ [name, out.dst, in.src, out.dst.name, in.src.name]. Similarity of *Place$* and *Place1$* is: 0.5555555555555556.

Table 3 shows the result of the evaluation on the large benchmark cases of [5].

Table 3. Evaluation on cases of [5]

Case	*Precision*	*Recall*	*F-measure*
EER2WebML	0.611	1.0	0.805
Ecore2EER	0.471	0.667	0.569
Ecore2WebML	0.476	0.526	0.501
UML1.4-EER	0.476	0.5	0.483
UML2.0-EER	0.563	0.667	0.615
UML1.4-WebML	0.714	0.462	0.588
UML2.0-WebML	0.733	0.556	0.645

Data of these and other cases may be found at nms.kcl.ac.uk/kevin.lano/mtsynthesis. The prototype tools used are at: nms.kcl.ac.uk/kevin.lano/uml2web.

7 Conclusion

In this paper, we have presented an approach for extracting correspondences by using metamodel matching. The core step has been calculating the similarities of all possible class matches, and using name similarity or human choice to select the best matching. Evaluation has shown the feasibility and relevance of the approach.

Our future research plan involves (i) compare different similarity measures for metamodel matching based on graph or data structures, naming, or semantics using several cases; (ii) compare different evolutionary approaches and other techniques for handling larger metamodels; (iii) develop a general approach for automatically transforming correspondences to transformations in QVT-R and other MT languages, and evaluate it with several examples to verify efficiency and quality. (iv) Finally, a tool will be developed based on our approach for automated model transformation.

References

1. Beyer, D., Lewerentz, C., Simon, F.: Impact of inheritance on metrics for size, coupling, and cohesion in object-oriented systems. In: International Workshop on Software Measurement. pp. 1–17. Springer (2000)
2. Bunke, H., Riesen, K.: Recent advances in graph-based pattern recognition with applications in document analysis. Pattern Recognition **44**(5), 1057–1067 (2011)
3. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Managing model adaptation by precise detection of metamodel changes. In: European Conference on Model Driven Architecture-Foundations and Applications. pp. 34–49. Springer (2009)
4. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Cope-automating coupled evolution of metamodels and models. In: European Conference on Object-Oriented Programming. pp. 52–76. Springer (2009)
5. Kessentini, M., Ouni, A., Langer, P., Wimmer, M., Bechikh, S.: Search-based metamodel matching with structural and syntactic measures. Journal of Systems and Software **97**, 1–14 (2014)
6. Kless, D., Milton, S.: Comparison of thesauri and ontologies from a semiotic perspective. In: Proc. of the 6th Australian Ontology Workshop (AOW 2010), T. Meyer, MA Orgun and K. Taylor (eds.) Adelaide, AU: Australian Computer Society. pp. 35–44. Citeseer (2010)
7. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady. vol. 10, pp. 707–710 (1966)
8. Macindoe, O., Richards, W.: Graph comparison using fine structure analysis. In: 2010 IEEE Second International Conference on Social Computing. pp. 193–200. IEEE (2010)
9. Maedche, A., Staab, S.: Comparing ontologies-similarity measures and a comparison study. AIFB (2001)
10. OMG: Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification. `http://www.omg.org/spec/QVT/1.1`, last accessed 17 Feb 2019
11. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: European Conference on Object-Oriented Programming. pp. 600–624. Springer (2007)
12. Wimmer, M., Langer, P.: A benchmark for model matching systems: The heterogeneous metamodel case. Softwaretechnik-Trends: Vol. 33, No. 2 (2013)