Automated Synthesis of ATL Transformations from Metamodel Correspondences

Kevin Lano\(^a\) and Shichao Fang\(^b\)

King’s College London, London, U.K.

Keywords: Model Transformations, ATL.

Abstract: In this paper we describe techniques for semi-automatically synthesising transformations from metamodel correspondences, in order to accelerate transformation development. We provide a strategy for synthesising complete ATL transformations from correspondences, and evaluate the approach using examples from the ATL zoo.

1 INTRODUCTION

Model transformations (MT) are central elements of model-driven engineering (MDE). However, MT specifications are often difficult to manually construct, and MT definition requires a high level of expertise in particular MT languages. The development of transformations can be a time-consuming process, which limits the agility of MDE. Therefore it is interesting to examine to what extent the transformation development process can be automated. Previous work has shown that it is possible to automatically recognise correspondences between source and target metamodels of a proposed transformation, and to use these as the basis of a transformation definition (Fang and Lano, 2019; Kessentini et al., 2014; Schwichtenberg et al., 2014).

In this paper we look in detail at how transformations can be constructed from the identified correspondences, and define a process for synthesising ATL transformations. We evaluate the benefits of the process in terms of the reduction of effort expended in MT construction, and in terms of the improved quality of the resulting transformations.

2 DERIVING TRANSFORMATION SPECIFICATIONS FROM METAMODELS

Transformations operate on source and target metamodels, usually one source metamodel \(MM_1\), and one target metamodel \(MM_2\). When a transformation developer is starting to construct a MT specification, their first task is usually to identify which classes and features of \(MM_1\) should be mapped to which classes and features of \(MM_2\). In other words, what the correspondence relation \(m\) (mapping) of \(MM_1\) and \(MM_2\) classes should be, and what the correspondence relation \(fm\) (feature mapping) of \(MM_1\) and \(MM_2\) features should be.

As an example, Figure 1 shows the metamodels of the ATL Class2Relational transformation case from the ATL zoo (www.eclipse.org/atl/atlTransformations). The source metamodel \(MM_1\) is \(Class\), on the LHS, the target metamodel \(MM_2\) is \(Relational\), on the RHS.

In this paper we have implemented different techniques for recognising \(m\) and \(fm\), based on structural, linguistic and semantic similarities of \(MM_1\) and \(MM_2\) elements (Fang and Lano, 2019). These have been defined as a plugin to the Eclipse Agile UML tools (projects.eclipse.org/projects/modeling.agileuml). Metamodels are imported as KM3 or Ecore files.

For example, the correspondence of \(Class\) to \(Table\) and of \(Attribute\) to \(Column\) in the above case are recognised by the structural similarity of these elements, whilst \(DataType\) in \(MM_1\) and \(Type\) in \(MM_2\)

---

\(a\) https://orcid.org/0000-0002-9706-1410  
\(b\) https://orcid.org/0000-0002-3556-4346
are linguistically similar. We primarily use structural similarity to identify class and feature correspondences, with linguistic and semantic similarity as secondary criteria.

To formalise class and feature correspondences, we use an abstract transformation notation called $\mathcal{T}L$. The notation expresses class correspondences of $E$ to $F$ by the notation $E \mapsto F$, and feature correspondences of $f$ to $g$ by the notation $f \mapsto g$.

A $\mathcal{T}L$ specification consists of a set of rules of the form

$$\{\text{PreCond}\} \ E \mapsto F$$

$$p_1 \mapsto q_1$$
$$\vdots$$
$$p_n \mapsto q_n$$

where the $p_i$ are features (owned, inherited or composed) of $E$, or OCL expressions in such features, and the $q_i$ are distinct features of $F$. The optional PreCond is a boolean-valued OCL expression in the $p_i$.

The meaning of a class mapping $\{C\} E \mapsto F$ as a model transformation rule is that for every instance $e$ of $E$ that satisfies $C$, there is a corresponding instance $e'$ of $F$. By default, distinct instances of $E$ map to distinct instances of $F$. A feature mapping $p \mapsto q$ of $\{C\} E \mapsto F$ means that for corresponding instances $e : E$, $e' : F$, the value of $e'.q$ is the interpretation $(e.p)'$ of $e.p$ via the class mappings.

For Class2Relational, the initial automatically-derived correspondences are:

- $\text{NamedElt} \mapsto \text{Name}$
  - name $\mapsto$ name
- $\text{Class} \mapsto \text{Table}$
  - name $\mapsto$ name
  - attr $\mapsto$ col
- $\text{Attribute} \mapsto \text{Column}$
  - name $\mapsto$ name
  - owner $\mapsto$ owner
  - type $\mapsto$ type
- $\text{Classifier} \mapsto \text{Type}$
  - name $\mapsto$ name
- $\text{DataType} \mapsto \text{Type}$
  - name $\mapsto$ name

Having obtained such possible correspondences, the next step of a MT specifier is typically to examine them for incompleteness or inconsistency. In the above example, we notice that the reference feature super of Class is not used by the discovered mappings (incompleteness), and in addition, there is a potential inconsistency in that Class is mapped to Table, but Table is not a specialisation of (or equal to) the image Type of Classifier, even though Class is a specialisation of Classifier.

Our tools partially automate this step by recognising cases of incompleteness and inconsistency, proposing solutions (additional or alternative mappings) to resolve them, and asking the user to confirm these solutions. Table 1 summarises the different checks which we use.

For the case of feature mapping incompleteness in Class2Relational, because the unused source feature super is a self-association on Class, the system proposes to replace $\text{attr} \mapsto \text{col}$ by the mapping

$\text{Set}\{\text{self}\} \mapsto \text{closure}\left(\text{super}$ $\mapsto \text{unionAll}\{\text{attr}\}\right) \mapsto \text{col}$

of all defined attributes of a class to the columns of a table, i.e., all attributes of the class itself and of all its ancestors are mapped to columns of the table corresponding to the class.

Because of the inheritance conflict in the targets of the class mappings, the additional class mapping

$\text{Class} \mapsto \text{Type}$

name $\mapsto$ name

is also proposed: this is a ‘vertical entity splitting’ of Class (Lano et al., 2018b): each Class instance in a source model is represented by both a Type instance and a Table instance in the resulting Relational model\(^1\).

\(^1\)The target classes must have no common $\text{MM2}$ super-class which is a type/element type of some $g \in \text{ran}(fm)$
Table 1: Consistency and completeness checks.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Correction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class mapping</td>
<td>Retarget Sub mapping, or add target splitting map for Sub subclass of E, has T not subclass/or equal to F, where E → F</td>
</tr>
<tr>
<td>Two directions of bidirectional association r not mapped to mutually reverse target features</td>
<td>Modify one feature mapping to ensure consistency or add for Sub subclass of E, target splitting map has T not subclass/or Sub ↦→ F equal to F, where E ↦→ F</td>
</tr>
<tr>
<td>Source, target features have different multiplicities</td>
<td>Propose modified mappings</td>
</tr>
<tr>
<td>Unused target subclasses F1 of F, where E → F</td>
<td>Introduce condition F1C and mapping {F1C}E ↦→ F1</td>
</tr>
<tr>
<td>Unused source or target feature f</td>
<td>Suggest class or feature mapping that uses f</td>
</tr>
<tr>
<td>Feature mapping f → r, g with r : R of abstract type/element type</td>
<td>Propose concrete subclass RSub of R for instantiation of r.</td>
</tr>
</tbody>
</table>

A more complex example is the SimpleClass2SimpleRDB case from the ATL zoo (Figure 2).

In this case there is incompleteness in the initial feature mapping, because it omits the target feature Table :: pkey and the source feature Attribute :: isPrimary. Because Class :: attrs is already mapped to a target feature, the system proposes the additional mapping

\[
\text{attrs} \rightarrow \text{select(isPrimary)} \rightarrow \text{pkey}
\]

of Class ↦→ Table. Further unused source boolean features can be used for class mapping conditions, eg., \{isPersistent\} Class ↦→ Table.

Another case of feature mapping incompleteness is when an association is navigated in opposite directions in the two metamodels, ie., there are matchings E → F, E1 → F1 of classes, but an association E → E1 in MM1 should correspond to the reverse direction rr' of an association F1 → F in MM2. To detect such cases, we look for unmatched source reference features E :: r where there is an unmatched target reference feature F1 :: rr whose (unnamed) reverse direction is type-compatible with r. Figure 3 shows a typical situation.

The expression matching

\[
E \rightarrow \text{select(ex | self = ex.r) } \rightarrow \text{rr}
\]

is then proposed as an additional E1 → F1 mapping, in the case that r is of 1-multiplicity. For ∗-multiplicity r the matching is

\[
E.allInstances() \rightarrow \text{select(}
\text{ex | ex.r.includes(self)) } \rightarrow \text{rr}
\]

For SimpleClass2SimpleRDB, such a case arises in the mapping of Association to FKey. Association ::
dest should be mapped to FKey :: references, and Association :: src should be mapped to the opposite direction of Table :: fkeys. The system recognises this and adds the additional mapping to Class → Table.

Feature mapping inconsistencies may arise because the two directions of bidirectional associations must be mapped in mutually consistent ways. If there is a mapping
\[ E \mapsto F \]
\[ r1 \mapsto rr1 \]
where \( r1 : E \rightarrow E1 \) has reverse reference \( r2 : E1 \rightarrow E \), and \( rr1 : F \mapsto F1 \) has reverse \( rr2 : F1 \rightarrow F \), then any mappings \( E1 \mapsto F1 \) or of subclasses \( E2 \) of \( E1 \) to \( F1 \) or to subclasses of \( F1 \) should include the feature mapping \( r2 \mapsto rr2 \). If instead a different mapping for \( r2 \) is present, the user is asked to resolve this conflict. This situation arises in the SQL to KM3 case, where the automatically derived correspondence Table → Class has columns → structuralFeatures, but Column → StructuralFeature has table → type instead of the correct mapping table → owner.

The SQL to KM3 case also illustrates a form of class mapping incompleteness, in which some concrete classes \( F1 \) (here, Reference and Attribute) are not the targets of any class mapping, but they have a superclass \( F \) in MM2 which is a target of a mapping \( E \mapsto F \) (here, Column → StructuralFeature). The system prompts the user to define more specific mappings
\[
\{ \text{Cond1} \} \ E \mapsto F1 \\
\{ \text{Cond2} \} \ E \mapsto F2
\]
for disjoint conditions Cond1, Cond2. This is the ‘horizontal entity splitting’ pattern of (Lano et al., 2018b). Unused boolean attributes of \( E \) are proposed as candidates for such discriminator conditions Cond1, Cond2, however the user can define their own conditions – in this example the conditions are the somewhat obscure \text{comment.size()} = 0 \text{ and its negation.}

A related feature mapping incompleteness issue arises with composed target features \( r.f \) where \( r \) is of an abstract class element type \( R \). In order to produce an executable implementation, some concrete subclass of \( R \) must be chosen as the type of \( r \).

Finally, some proposed class mappings may be recognised as spurious and unnecessary, if there is no feature mapping \( p \mapsto q \) which uses the class mapping.

In general, the process of refining correspondences based on identified incompleteness and inconsistencies can simulate the decisions which a human MT expert might make. Indeed the automated detection of all such issues can ensure that many potential semantic gaps and conflicts in a transformation specification are identified and resolved prior to implementation. A purely manual specification process could result in some issues being missed.

\section{MAPPING FROM \( \mathcal{T} \mathcal{L} \) TO ATL}

Once a complete and consistent \( \mathcal{T} \mathcal{L} \) specification is obtained, the next step is to generate an implementation of this in a particular MT language, such as QVTr, QVTo, UML-RSDS, ETL or ATL.

We describe how QVT implementations of \( \mathcal{T} \mathcal{L} \) transformations can be derived in (Fang and Lano, 2019). Because ATL (Eclipse, 2019) is a more imperative language than QVTr, and also more restricted in its declarative part than QVT, the mapping of \( \mathcal{T} \mathcal{L} \) specifications to ATL is more complex. In this section we describe the steps of this mapping.

For each \( \mathcal{T} \mathcal{L} \) rule \{ Cond \} \( E \mapsto F \), with \( F \) a concrete class, there is an ATL matched rule of the schematic form

\begin{verbatim}
rule E2F
  { from ex : MM1!E ( ex.Cond )
  to fx : MM2!F
  { ... }
}
\end{verbatim}

In the case that there are two or more \( \mathcal{T} \mathcal{L} \) rules for \( E \) with the same conditions Cond, these must be combined into a single ATL rule with multiple output pattern elements.

For example, rules \( E \mapsto F1 \) and \( E \mapsto F2 \) with concrete \( F1, F2 \) would be implemented by:

\begin{verbatim}
rule E2F1F2
  { from ex : MM1!E
  to f1x : MM2!F1
  { ... },
  f2x : MM2!F2
  { ... }
}
\end{verbatim}

This means also that source expressions \( e \) of element type \( E \) must be disambiguated when used on the RHS of bindings: \( t \leftarrow e \) refers to \( e \) converted to \( F1 \) elements via \( E2F1F2 \), whilst \( t \leftarrow thisModule.resolveTemp(e,'f2x') \) refers to \( e \) converted to \( F2 \) elements via the rule.

A feature mapping \( f \mapsto g \) is represented by an ATL binding \( g \leftarrow ex.f \) in cases where \( f \) and \( g \) are not composed. Source compositions \( r.f \) for \( f \) of 1 multiplicity, \( r \) of \(*\) multiplicity, are evaluated as \( ex.r\rightarrow collect( \lambda x. x.f ) \).
In the case of a feature mapping \( f \mapsto r.g \), where \( r \) is of 1-multiplicity, of concrete class element type \( R \), a new output pattern element \( rx : MM2!R \) is defined:

\[
\text{rule } E2F \quad \{ \text{ from } ex : MM1!E \ (\text{ex.Cond}) \to fx : MM2!F \ (r \leftarrow rx) \}
\]

The bindings for \( rx \) implement the feature mappings for \( f \mapsto g \), and for other mappings \( k \mapsto l \), for each \( k \mapsto r.l \) which is a feature mapping of the \( E \mapsto F \) mapping.

In the case that there is a direct mapping \( f \mapsto r \) and also composed mappings \( g \mapsto r.h \), a do clause is needed for the composed mappings:

\[
\text{rule } E2F \quad \{ \text{ from } ex : MM1!E \ (\text{ex.Cond}) \to fx : MM2!F \ (r \leftarrow ex.f) \}
\]

where \( r \) is of 1 multiplicity.

For \( 0..1 \) and for *-multiplicity \( r \), a do clause implementation of \( g \mapsto r.h \) instead has the form

\[
\text{for } (rx \in fx.r) \{ rx.l \leftarrow ex.k; \}
\]

If there are cases of mappings \( f \mapsto r.g \) where \( f \) and \( r \) are of *-multiplicity, then instead sets of \( R \) elements are created using unique lazy or called rules. In this case, if \( f \) has a class element type \( E1 \), the mapping \( f \mapsto r.g \) in cases where \( r \) and \( f \) have *-multiplicity can be implemented by introducing a new unique lazy rule:

\[
\text{rule } E2F \quad \{ \text{ from } ex : MM1!E \ (\text{ex.Cond}) \to fx : MM2!F \ (r \leftarrow ex.f->collect( elx | thisModule.MapE12Rg(elx) )) \}
\]

unique lazy rule MapE12Rg
\[
\text{from elx : MM1!E1 to rx : MM2!R} \quad \{ g \leftarrow elx \}
\]

The effect of this approach is to produce a set of \( R \) objects, one for each element \( elx \) of \( ex.f \). Again, \( R \) must be a concrete class for this to be valid. An example of this situation is the supertypes \( \mapsto \) generalization.parent mapping in the MOF2UML case.

If \( f \) is of \( 0..1 \) multiplicity and \( r \) of * or \( 0..1 \) multiplicity, then the assignment to \( r \) is

\[
r \leftarrow \text{Set}(ex.f)->collect( elx | thisModule.MapE12Rg(elx) )
\]

Further updates to the \( r \) with additional mappings \( k \mapsto r.l \) with \( l \) of higher or equal upper multiplicity to \( k \) must be handled in the do clause of the \( E2F \) rule, via a statement

\[
\text{for } (rx \in fx.r) \{ rx.l \leftarrow ex.k; \}
\]

Further updates to the \( r \) with additional mappings \( k \mapsto r.l \) with \( l \) of smaller upper multiplicity than \( k \) require the creation of additional \( R \) objects:

\[
r \leftarrow ex.k->collect( e2x | thisModule.MapE22Rl(e2x) )
\]

If instead \( f \) has a value element type \( T \), the mapping \( f \mapsto r.g \) can be implemented by introducing a new called rule:

\[
\text{rule } E2F \quad \{ \text{ from } ex : MM1!E \ (\text{ex.Cond}) \to fx : MM2!F \ (r \leftarrow ex.f->collect( tx | thisModule.MapT2Rg(tx) )) \}
\]

\[
\text{rule MapT2Rg(tx : T) : MM2!R} \quad \{ to rx : MM2!R \ (g \leftarrow tx) \}
\]

Table 2 summarises the translation from \( I \L \) to ATL for composed target feature mappings \( f \mapsto r.g \), where there is no direct mapping \( h \mapsto r \).

The synthesised ATL satisfies the quality recommendations of (Eclipse, 2019):

- Use standard and unique lazy rules in preference to lazy rules – lazy rules are not needed by our translation.
- Avoid imperative constructs and the use of resolveTemp – imperative code and called rules are only needed in cases of composite target features. resolveTemp is only needed in cases of vertical entity splitting.
- Use collect in preference to iterate – iterate is only needed to define the closure operator, otherwise collect is used.

More precisely, the situation with regard to the quality flaws of (Lano et al., 2018a) is that cycles of calling dependencies (\( CBR_2 > 0 \)) are not possible, while errors of excessive fan-in, fan-out and
Table 2: Generated ATL for composite target feature mappings $f \rightarrow r.g$ of $E \rightarrow F$.

<table>
<thead>
<tr>
<th>$r$ multiplicity</th>
<th>$f$ multiplicity</th>
<th>$g$ multiplicity</th>
<th>ATL implementation</th>
</tr>
</thead>
</table>
| 1                | any              | any              | $r \leftarrow rx$ for new OutPatternElement $rx : MM2!R (\ldots)$
|                  |                  |                  | $rx$ binding is |
|                  |                  |                  | $g \leftarrow ex.f$ for $g$ upper multiplicity $\geq f$ upper multiplicity, |
|                  |                  |                  | $g \leftarrow ex.f \rightarrow \text{any}()$ otherwise. |
|                  |                  |                  | All bindings due to $E \rightarrow F$ mappings $k \rightarrow r.l$ |
|                  |                  |                  | are defined in the $rx$ pattern. |
| *                | 1                | any              | $r \leftarrow rx$ for new OutPatternElement $rx : MM2!R (\ldots)$
|                  |                  |                  | $rx$ binding is $g \leftarrow ex.f$ |
| * or 0..1        | 0..1             | any              | $r \leftarrow \text{Set}\{ex.f\} \rightarrow \text{collect}(ex1 | thisModule.MapE12Rg(ex1))$ |
|                  |                  |                  | Additional $k \rightarrow r.l$ with |
|                  |                  |                  | upper multiplicity $k \leq$ upper multiplicity $l$: |
|                  |                  |                  | for $(rx \in fx,r)$ \{ $rx.l \leftarrow \text{ex.k;}$ \} |
|                  |                  |                  | Additional $k \rightarrow r.l$ with |
|                  |                  |                  | upper multiplicity $k >$ upper multiplicity $l$: |
|                  |                  |                  | $r \leftarrow ex.k \rightarrow \text{collect}(ex2 | thisModule.MapE22Rl(ex2))$ |
| *                | *                | any              | $r \leftarrow \text{ex.f} \rightarrow \text{collect}(ex1 | thisModule.MapE12Rg(ex1))$ |
|                  |                  |                  | Additional $k \rightarrow r.l$ are treated as for $0..1$ multiplicity |

Calling dependencies can arise only because of calls to data conversion operators or to auxiliary unique lazy/called rules in the case of composite target features. Duplicate code may arise if two subclasses $E_1$, $E_2$ of a source class $E$ have common feature mappings – this duplication could be removed by using ATL rule inheritance. Excessive rule and transformation size may occur due to the size of the metamodels.

4 EVALUATION

We evaluated the approach using several cases from the ATL transformation zoo. For these cases we generated a solution using only the provided metamodels in KM3 or Ecore format. The procedure detailed in Section 2 was followed with interactive enhancement of the initially derived metamodel correspondences. In Table 3 we compare our derived ATL transformation with the manually-constructed solutions of the zoo cases, in terms of their size and the number of technical debt flaws in the transformation according to the quality flaw categories of (Lano et al., 2018a). We also quantify the recall, precision and f-measure of our matching result with respect to the class and feature mappings defined by the original manually-created transformations. In terms of quality, the synthesised transformations have lower numbers of flaws, and lower flaw density (the average flaw density of the generated transformation versions is 0.008 flaws/LOC, compared to 0.0172 for the original versions). All the examples are available at https://nms.kcl.ac.uk/kevin.lano/mtsynthesis.

Table 4 shows some estimates of the relative amount of work involved in the manual and automated construction of the ATL zoo case versions. We estimate effort in terms of how many manual changes are necessary to the automatically-synthesised class and feature mappings (additional to changes identified by the interactive improvement process), and the semantic complexity of the original versions, as the total number of class and feature mappings. The execution time for the automated synthesis of the initial $TL$ specifications is also shown (times are the same for a transformation and its inverse because our tool generates these together).

These results show that a relatively small amount (less than 10% of the feature/class mappings) of transformation content needs to be manually modified or created, for the versions produced by our transformation synthesis process.

5 RELATED WORK

For metamodel matching, there are established benchmarks cases, and results of different approaches on these benchmarks are given in (Addazi et al., 2016; Fang and Lano, 2019; Kessentini et al., 2014; Voigt and Heinze, 2010). We show in Table 5 that our meta-
Table 3: Evaluation on ATL zoo cases.

<table>
<thead>
<tr>
<th>Case</th>
<th>Original size (LOC)</th>
<th>Original flaws</th>
<th>Recall</th>
<th>Precision</th>
<th>F-measure</th>
<th>New size (LOC)</th>
<th>New flaws</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ports</td>
<td>31</td>
<td>0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>PetriNet2PathExp</td>
<td>70</td>
<td>1</td>
<td>0.78</td>
<td>1.0</td>
<td>0.875</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>Class2Relational</td>
<td>97</td>
<td>2</td>
<td>1.0</td>
<td>0.88</td>
<td>0.94</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>PathExp2PetriNet</td>
<td>104</td>
<td>0</td>
<td>0.94</td>
<td>1.0</td>
<td>0.97</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>SimpleClass2</td>
<td>302</td>
<td>10</td>
<td>1.0</td>
<td>0.87</td>
<td>0.93</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td>SimpleRDB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ant2Maven</td>
<td>324</td>
<td>4</td>
<td>0.84</td>
<td>0.99</td>
<td>0.91</td>
<td>317</td>
<td>2</td>
</tr>
<tr>
<td>Maven2Ant</td>
<td>360</td>
<td>3</td>
<td>0.92</td>
<td>0.91</td>
<td>0.91</td>
<td>332</td>
<td>2</td>
</tr>
<tr>
<td>MOF2UML</td>
<td>585</td>
<td>8</td>
<td>0.69</td>
<td>0.78</td>
<td>0.73</td>
<td>255</td>
<td>3</td>
</tr>
<tr>
<td>MySQIL2KM3</td>
<td>613</td>
<td>13</td>
<td>0.79</td>
<td>0.81</td>
<td>0.8</td>
<td>48</td>
<td>1</td>
</tr>
<tr>
<td>UML2MOF</td>
<td>935</td>
<td>18</td>
<td>0.53</td>
<td>0.8</td>
<td>0.64</td>
<td>220</td>
<td>3</td>
</tr>
<tr>
<td>Averages</td>
<td>342.1</td>
<td>5.9</td>
<td>0.85</td>
<td>0.9</td>
<td>0.87</td>
<td>134</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Table 4: Effort of manual/automated versions of cases.

<table>
<thead>
<tr>
<th>Case</th>
<th>Execution time (ms)</th>
<th>Changed maps</th>
<th>Original maps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ports</td>
<td>70</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>PetriNet2PathExp</td>
<td>133</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Class2Relational</td>
<td>60</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>PathExp2PetriNet</td>
<td>133</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>SimpleClass2</td>
<td>285</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>SimpleRDB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ant2Maven</td>
<td>326,769</td>
<td>1</td>
<td>136</td>
</tr>
<tr>
<td>Maven2Ant</td>
<td>326,769</td>
<td>7</td>
<td>112</td>
</tr>
<tr>
<td>MOF2UML</td>
<td>66,953</td>
<td>32</td>
<td>184</td>
</tr>
<tr>
<td>MySQIL2KM3</td>
<td>859</td>
<td>5</td>
<td>128</td>
</tr>
<tr>
<td>UML2MOF</td>
<td>66,953</td>
<td>31</td>
<td>165</td>
</tr>
</tbody>
</table>

The automated synthesis approach produces similar overall results to the SBSE approach of (Kessentini et al., 2014) on the benchmarks, and therefore is superior to the results of other approaches with regard to these benchmarks, such as (Addazi et al., 2016; Voigt and Heinze, 2010). In terms of efficiency, the execution times on the benchmarks are generally lower than the results of (Kessentini et al., 2014), although we use deterministic algorithms instead of the evolutionary algorithm approach of (Kessentini et al., 2014).

Several existing works on recognising metamodel correspondences are based on the concept of similarity flooding (Melnik et al., 2002; Addazi et al., 2016; Fabro and Valduriez, 2009; Garces et al., 2009): that if two metamodel elements $c_1$, $c_2$ (e.g., classes) are connected to elements $e_1$, $e_2$ with a known similarity $\nu$, then $c_1$ and $c_2$'s estimated similarity can be modified based on $\nu$ and the strength of the connections. However this approach only takes into account pairwise similarities, whilst in practice many-to-many similarities may exist between metamodels (e.g., a group of classes in $MM_1$ can be related to a group in $MM_2$). Thus our matching approach considers global and n-to-m matches of metamodel elements as a basis for metamodel correspondences. A similar global structural matching approach using graph similarity is described in (Voigt and Heinze, 2010), however they rely on an initial manually-constructed ‘seed’ matching of classes, and on planarization of metamodel graphs, which our approach does not need. In this paper and in (Fang and Lano, 2019) we consider 15 of the 20 ‘gold standard’ cases of (Voigt and Heinze, 2010), and we achieve an average F-score of 0.79 on these, compared to 0.58 in (Voigt and Heinze, 2010).

Other approaches to synthesising transformations from correspondences are (Fabro and Valduriez, 2009) and (Garces et al., 2009). (Fabro and Valduriez, 2009) defines case-specific patterns to create transformations for particular source and target metamodels, however we define general-purpose patterns and consistency and completeness checks to identify and refine correspondences for arbitrary metamodel pairs. For example, the mutual consistency of feature mappings of the two directions of a bidirectional association is a logical property which must hold for any semantically-valid transformation, and hence a proposed mapping $m$, $fm$ must satisfy this property or be modified to satisfy it. (Fabro and Valduriez, 2009) does not consider the quality of generated transformations, and its ATL generation approach does not appear to address the issue of composed target features, which considerably complicates ATL production.

The AMW tool of (Garces et al., 2009) provides a means to define customised matching criteria and techniques. This requires more substantial intervention from the developer than our approach, which is
Table 5: Evaluation on cases of (Kessentini et al., 2014).

<table>
<thead>
<tr>
<th>Case</th>
<th>Precision (SBSE)</th>
<th>Recall (SBSE)</th>
<th>F (SBSE)</th>
<th>Precision (SBSE)</th>
<th>Recall (SBSE)</th>
<th>F (SBSE)</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebML2EER</td>
<td>0.65</td>
<td>1.0</td>
<td>0.79</td>
<td>0.69</td>
<td>0.72</td>
<td>0.70</td>
<td>0.116</td>
</tr>
<tr>
<td>EER2Ecore</td>
<td>0.6</td>
<td>1.0</td>
<td>0.75</td>
<td>0.48</td>
<td>0.59</td>
<td>0.52</td>
<td>0.188</td>
</tr>
<tr>
<td>WebML2Ecore</td>
<td>0.7</td>
<td>0.74</td>
<td>0.72</td>
<td>0.82</td>
<td>0.69</td>
<td>0.74</td>
<td>0.125</td>
</tr>
<tr>
<td>EER2UML1.4</td>
<td>0.8</td>
<td>0.86</td>
<td>0.83</td>
<td>0.71</td>
<td>0.72</td>
<td>0.71</td>
<td>0.172</td>
</tr>
<tr>
<td>EER2UML2.0</td>
<td>0.61</td>
<td>0.67</td>
<td>0.64</td>
<td>0.67</td>
<td>0.72</td>
<td>0.68</td>
<td>0.422</td>
</tr>
<tr>
<td>WebML2UML1.4</td>
<td>0.53</td>
<td>0.62</td>
<td>0.57</td>
<td>1.0</td>
<td>0.84</td>
<td>0.91</td>
<td>0.203</td>
</tr>
<tr>
<td>WebML2UML2.0</td>
<td>0.79</td>
<td>0.75</td>
<td>0.77</td>
<td>0.91</td>
<td>0.73</td>
<td>0.81</td>
<td>0.219</td>
</tr>
<tr>
<td>Ecore-UML1.4</td>
<td>0.76</td>
<td>0.64</td>
<td>0.69</td>
<td>1.0</td>
<td>0.66</td>
<td>0.79</td>
<td>104.5</td>
</tr>
<tr>
<td>Ecore-UML2.0</td>
<td>0.76</td>
<td>0.78</td>
<td>0.77</td>
<td>0.64</td>
<td>0.89</td>
<td>0.75</td>
<td>510</td>
</tr>
<tr>
<td>UML1.4-UML2.0</td>
<td>0.9</td>
<td>0.6</td>
<td>0.72</td>
<td>1.0</td>
<td>0.67</td>
<td>0.8</td>
<td>393</td>
</tr>
<tr>
<td>Average</td>
<td>0.71</td>
<td>0.77</td>
<td>0.73</td>
<td>0.79</td>
<td>0.72</td>
<td>0.74</td>
<td></td>
</tr>
</tbody>
</table>

primarily automated and requires relatively low manual intervention. It is unclear if AMW is able to generate complex ATL for cases of composed target features.

Transformation construction by example is another approach for semi-automated transformation derivation (Balogh and Varro, 2008). Transformation rules are inferred from examples of the intended transformation inputs and outputs (models). Example source and target models could also be used to enhance our approach, to identify detailed feature mappings which cannot be inferred from feature typing (e.g., \(2 \times x \rightarrow y\) for integer-typed features \(x\) and \(y\)).

6 CONCLUSIONS

We have described a process for synthesising ATL transformations from metamodel correspondences, based on analysis of the consistency and completeness of these correspondences. The approach is novel in attempting to formally emulate the processes which a software engineer would informally undertake when creating a transformation.

We have shown that this approach can produce correct and effective transformations, with a higher quality than manually-produced transformation code, and that development times for transformations can be reduced in principle by the approach.

REFERENCES


