Unit Testing of Model to Text Transformations

Alessandro Tiso, Gianna Reggio, Maurizio Leotta

DIBRIS – Università di Genova, Italy
alessandro.tiso | gianna.reggio | maurizio.leotta@unige.it

Abstract. Assuring the quality of Model Transformations, the core of Model Driven Development, requires to solve novel and challenging problems. Indeed, testing a model transformation could require, for instance, to deal with a complex software artifact, which takes as input UML models describing Java applications and produces the executable source code for those applications. In this context, even creating the test cases input and checking the correctness of the output provided by the model transformation are not easy tasks. In this work, we focus on Model to Text Transformations and propose a general approach for testing them at the unit level. A case study concerning the unit testing of a transformation from UML models of ontologies into OWL code is also presented.

1 Introduction

Model to Text Transformations (shortly M2TTs) are not only used in the last steps of a complete model driven software development process to produce the code and the configuration files defining a new software system, but can be the way to allow any kind of user to perform tasks of different nature using visual models instead of the scarcely readable text required by software tools (e.g. the textual input for a simulator of business processes may be generated by a UML model made of class and activity diagrams).

Testing model transformations is a complex task [6], since the complexity of the inputs and the time required to produce them (e.g. complete UML models instead of numbers and strings), and the difficulties in building an effective oracle (e.g. it may have to provide whole Java programs instead of a result of such programs). Selecting input models for model transformations testing is harder than selecting input for programs testing because they are more difficult to be defined in an effective way. Hence, also giving adequacy criteria for model transformations is again more difficult than in the case of programs. Furthermore, also the well-established kinds of tests, e.g. acceptance and system, are either meaningless for model transformations or have to be redefined.

M2TTs may be quite complex and large. For example, a transformation from UML models composed of class diagrams and state machines, where the behaviour of any element is fully defined by actions and constraints, to running Java applications built using several frameworks (e.g. Hibernate, Spring, JPA). Thus, it is important to be able to perform tests also on subparts of an M2TT. First, we need to identify the nature of such subparts, and then define what testing them means. Using the classical terminology, we need to understand whether a form of unit testing is possible for M2TTs.

In this paper we present a proposal for testing M2TTs at the unit level. We then indicate how to implement the executions of the introduced unit tests for M2TTs coded using Acceleo [1], and how this form of unit testing may be integrated in the development method for model to text transformation *MeDMoT* proposed by some of the authors in [10, 11, 12].

We use the M2TT U-OWL as running example to illustrate our proposal. It is a transformation from profiled UML models of ontologies to OWL definitions of ontologies. An ontology model is composed by a class diagram (the StaticView) defining the structure of the ontology in terms of categories (plus specializations and associations among them), and possibly by an object diagram (the InstancesView) describing the information about the instances of the ontology (i.e. which are the individuals/particulars of the various categories, the values of their attributes, and the links among them). The U-OWL targets are text files describing an ontology using the RDF/XML for OWL concrete syntax. This transformation has been developed following *MeDMoT*. Specifically, we have given the U-OWL requirements, designed the transformation and then implemented it using Acceleo. It is composed by 8 modules, 21 templates and 8 queries.

The paper is structured as follows. In Sect. 2 we present our proposal for M2TTs unit testing including suggestions for implementing their execution and getting a report, and in Sect. 3 how this kind of testing may be integrated within *MeDMoT*. The results of the unit testing of U-OWL are summarized in Sect. 4. Finally, related work and conclusions are respectively in Sect. 5 and 6.

2 Unit Testing of Model to Text Transformations

2.1 Testing Model to Text Transformations

The *Model to Text Transformations* considered in this paper map models[1] (both graphical and textual) to *Structured Textual Artifacts* (shortly STAs) having a specific form. An STA is a set of text files, written using one or more concrete syntaxes, disposed in a well-defined structure (physical positions of the files in the file system).

In previous works [10, 11, 12] we have proposed new kinds of tests suitable for model to text transformations.

– **Conformance tests** are made with the intent of verifying whether the M2TT results comply the requirements imposed by the target definition. Generally this means that the produced STA has the required structure and form. Considering the U-OWL case this means that the files produced can be loaded into a tool for OWL like Protege[2] without errors (because it accepts only well-formed OWL files).
– **Semantic tests** are made with the intent of verifying whether the target of the M2TT has the expected semantics. In the U-OWL case a semantic test may check, for example, if an OWL class has all the individuals represented in the UML model, or that an OWL class is a sub-class of another one iff such relationship was present in the UML model.
– **Textual tests** are made with the intent of verifying whether the STAs produced by the M2TT have the required form considering both the structuring in folders and files and the textual content of the files.

[1] conform to a meta-model

[2] `http://protege.stanford.edu/`

2.2 Unit Tests for Model to Text Transformations

M2TTs may be quite large and complex. For this reason, they may be composed of several sub-transformations. Each sub-transformation may be in turn composed of other sub-transformations, each of them taking care of transforming some model elements. The various sub-transformations may be arranged in a call-graph (we use a diagram similar to the structure chart, that we call *functional decomposition diagram*), where the nodes are labelled by the sub-transformations themselves and the arcs represent calls between sub-transformations (see, e.g. Fig. 1).

In what follows, we consider only M2TTs equipped with a functional decomposition diagram; moreover, we assume that their design specifications, which provide information on the STAs produced by their sub-transformations, are available.

To perform the unit testing of an M2TT we must identify the units composing it. That will be the subjects of the unit tests. It is natural to choose as subject of unit tests the various sub-transformations of the M2TT, introduced by the functional decomposition diagram, considered in isolation.

In the context of testing classical software, the parts considered by unit tests, in general, cannot be run (e.g. a method or a procedure), thus special stubs have to be built (e.g. JUnit test cases for Java class methods). In general, in the case of M2TTs the "parts" composing a transformation return STA fragments that in many cases do not even correspond to well-formed constructs in the transformation target; to build some stubs for them to be able to perform conformance and semantic testing may be utterly complex and time consuming (think for example what does it mean to build a stub for a configuration file for the Spring framework, or referring to the U-OWL case study, to build a stub for an RDF/XML definition of an individual). To propose a general technique for unit testing of M2TTs, we should consider the sub-transformation results as purely textual artifacts, and so we use only textual tests, which is one of the proposed new kinds of tests (see Sect. 2.1).

To build a unit test for a given sub-transformation, say *ST*, we need other ingredients: the test input and the oracle function.

The *test inputs* for *ST* are model elements. For example, the sub-transformation of U-OWL **TAttribute** (see Fig. 1) transforms a UML class attribute into an OWL fragment

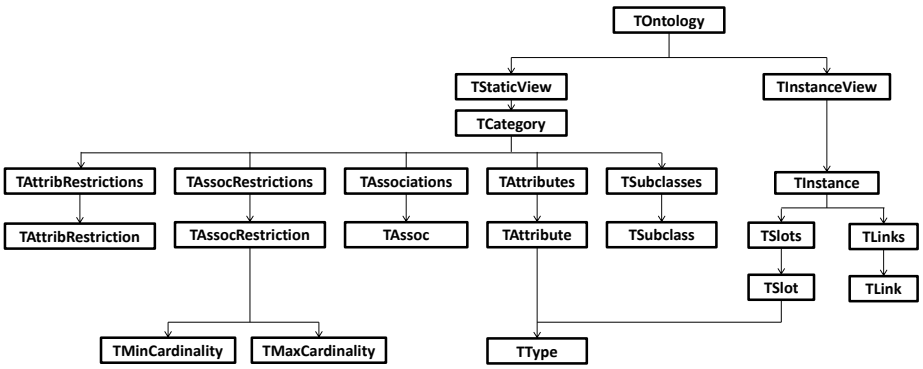

Fig. 1. Functional decomposition diagram for the U-OWL M2TT

(more precisely, in the definition of an OWL data type property). Thus, to test **TAttribute** we need to use as inputs UML attributes.

There are substantially two ways to obtain the model elements to be used in the unit tests: (i) they can be built by instantiating the right meta-class (in the case **TAttribute**, the meta-class in the UML meta-model which represents class attributes) or (ii) they can be extracted from a model using an appropriate query. We opted for (ii) because in this way input models can be easily written by the tester. Furthermore, if the model considered contains more than one model element of the desired type (e.g. class attribute), it is possible to easily improve the test coverage of the input space. Indeed, we can retrieve all these elements in only one query, and thus test the sub-transformation using different values.

The *oracle function* has to evaluate the correctness of the result of *ST* on the chosen input. Usually, this is done by comparing the actual output given by the *ST* with the expected output. In the case of the M2TTs the expected output of a sub-transformation is a fragment of an STA (for example some text files arranged in different folders, a text file, a text fragment). It is clear that generating this kind of expected outputs is very difficult and time consuming if not almost impossible; consider the case of the transformation from UML models to Java applications, the expected outputs may be sets of files containing the complete code of various Java classes or long and cryptic configuration files needed by the used framework (e.g. Hibernate).

Thus, we formulate the oracles in terms of properties on the expected STA fragments. These properties are about the structure of the STA fragment, for example which files and folders must be present and how they are named, and about the text files content. The latter may just require that the output text matches a given regular expression.

Following the classical testing techniques, we define a test case for a sub-transformation as a pair consisting of a specific input and a property on the expected output; we have preferred instead to define generic test cases that are properties on the expected output parameterized on the input of *ST*. To allow the expression of more powerful tests, we then decided that an oracle is composed by one or more conditions on the *ST* parameters that allow to select the proper parameterized regular expression for checking the correctness of the *ST* output.

A test suite is composed by a set of input models and by a list of test specifications (written by the tester), one for each *ST*, such as the one shown in Fig. 2. Obviously, at each test specification corresponds a test case in the test suite implementation. For simplicity in this case, we adopted a simplified regular expression language in which

Sub-Transformation

TAttribute(*attrName:*String,*OwnerClassName:*String)

Verification Pattern

```
<owl:DatatypeProperty rdf:ID = 'attrName'>
  <rdfs:domain rdf:resource = '#OwnerClassName'/>
  <rdfs:range rdf:resource = ***/>
</owl:DatatypeProperty>
```

Fig. 2. Unit Test Specification for **TAttribute**

the text that must be ignored is marked by a string containing three "*" characters (see Fig. 2).

2.3 Unit Tests Implementation

In what follows we assume that the considered M2TTs (and their sub-transformations) are implemented using the *Eclipse Platform* [4], taking advantage of the features offered by the *Eclipse Modeling Project* [3], such as:
– the *Eclipse Modeling Framework* [2] (EMF) that provides tools and runtime support for viewing and editing models, and
– Acceleo [1] that is an implementation of the OMG MOF Model to Text Language [5] plus an IDE and a set of tools to simplify the production of M2TTs.

Each M2TT is implemented as an Eclipse plugin, and it is composed by a set of Acceleo templates, which correspond to the sub-transformations appearing in the functional decomposition diagram.

The test suite is implemented as an Acceleo transformation, built by defining a template for each test case (thus, it is an Eclipse plugin). Obviously, the Eclipse plugin implementing the unit test suite must have a dependency to the plugin which implements the M2TT, so that a test case for *ST* can invoke the template corresponding to *ST*.

Let *TC* be the test case for the sub-transformation *ST*. The template implementing *TC* is defined as follows.

Fig. 3 shows the four logical steps performed by the Acceleo transformation implementing the test suite during its execution. First, an OCL query over the input model extracts the model elements suitable to be used as input for *ST* (e.g. **TAttribute**, see point

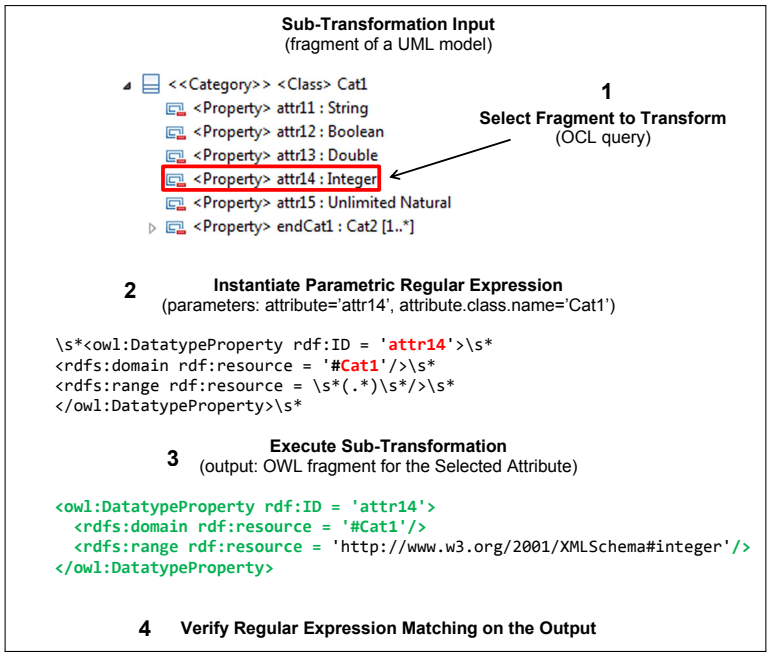

Fig. 3. Unit Test Logical Schema

Test Name: TestTTAttribute		
Function under test: TAttribute		
Model Element	Test Result	Text Fragment
Attribute: attr11 of Category: Cat1	OK	
Attribute: attr12 of Category: Cat1	OK	
Attribute: attr13 of Category: Cat1	OK	
Attribute: attr14 of Category: Cat1	OK	
Attribute: attr15 of Category: Cat1	OK	

Fig. 4. HTML Unit Test Report: no failed tests

Test Name: TestTTCategory		
Function under test: TCategory		
Model Element	Test Result	Text Fragment
Category: Cat1	FAIL	text fragment
Category: Cat10	FAIL	text fragment
Category: Cat11	FAIL	text fragment
Category: Cat12	FAIL	text fragment
Category: Cat2	FAIL	text fragment

Fig. 5. HTML Unit Test Report: failed tests

1 in Fig. 3). Then, for each model element, a special kind of Acceleo query, called Java Services wrapper[3], is invoked. This kind of Acceleo query can call a Java method as if it was a simple query. The service wrapper implements the oracle of *TC*. It takes the following parameters: a string obtained by invoking *ST*, that is the actual output of the *ST*, and the same parameters (model elements) used by *ST*. The service wrapper, using the parameters representing the model element, builds the appropriate regular expression (point **2** of Fig. 3, the parametric portion of the regular expression is depicted as red text) by instantiating the general parametric regular expression appearing in *TC* (see, e.g. Fig. 2). Then, the service wrapper, using the instantiated regular expression, checks if the output of *ST* (point **3**) is correct (point **4**).

Fig. 3 bottom, shows the actual output of the **TAttribute** sub-transformation and the green text represents the correct match of the regular expression shown above while black text represents text that is not checked (e.g. in the considered case study, the type of the attributes is managed by the **TType** sub-transformation and thus is not checked during the unit testing of the **TAttribute** sub-transformation).

The result of the oracle function is then used to build a HTML report as it is shown in Fig. 4 and 5. In detail, Fig. 4 shows a report of a unit test over a *ST* successful on all the elements, meanwhile Fig. 5 shows a report of a unit test failing on some elements. In the latter case on the rightmost column there is a hyper-link to the text fragment generated by the sub-transformation under test. Thus, the developer can inspect the problematic output.

Summarizing, each unit test, which has as subject one of the sub-transformations: (1) selects the model elements composing the test input using appropriate queries over the input models, and for each model element (2) calls the oracle function that, invokes the sub-transformation under test obtaining the actual output that is verified using a regular expression built in conformance to what is defined by the M2TT design specification for the sub-transformation parameterized with the actual values of the sub-transformation parameters. As last step, using the oracle function result, we produce a report of the unit test, in a form suitable to be easily readable from the transformation developer, such as a HTML report.

Obviously, the plugin containing the test suite can be executed using different models as input.[4]

[3] see `http://www.obeonetwork.com/page/acceleo-user-guide`

[4] It can be done by configuring in an appropriate way Eclipse run-configurations.

3 Model to Text Transformations Unit Testing within *MeDMoT*

In the previous section we described a general approach for testing at the unit level an M2TT. This approach can be integrated within *MeDMoT*, our general method for developing M2TTs. Each instance of *MeDMoT* is a method to develop a model to text transformations (e.g. U-OWL). It is briefly described in [11, 12], and more details can be found in the Ph.D. thesis of one of the authors [10]. Here, we summarize the main concepts and phases of *MeDMoT*.

The input of a transformation (e.g. U-OWL) is a set of UML models conform to the UML meta-model extended with a specific profile (e.g. a profile for UML models of ontologies). The form of the source models is defined by a conceptual meta-model, i.e. a UML class diagram with a class whose instances correspond to all the possible source models. That meta-model is constrained by a set of well-formedness rules that precisely define the set of acceptable models. The target of a transformation is always an STA having a specific form (e.g. for U-OWL a set of text files describing an ontology).

In *MeDMoT* an M2TT is structured as a chain of transformations of different types, some from model to model and one from model to text. The input model is verified by a model to model transformation that checks if the input model satisfies the well-formedness rules constraining the transformation source (i.e. the input model). If the input model is well-formed, then it is refactored by one or more model to model transformations in an equivalent simplified model. This step helps to maintain the subsequent and last M2TT as simple as possible, for instance, by reducing the number of different constructs used in the model. Finally, the refactored model is transformed into an STA using an M2TT. We call the last transformation of this chain the Text Generation Transformation (shortly TGT), which is precisely the M2TT on which we perform the unit tests.

The design of each sub-transformation of the TGT is specified by means of relevant source-target pairs. Each pair is composed by a left side, that shows a template for the sub-transformation inputs, and a right side that shows the result of the application of this sub-transformation on model elements obtained instantiating such template, see e.g. Fig. 6.

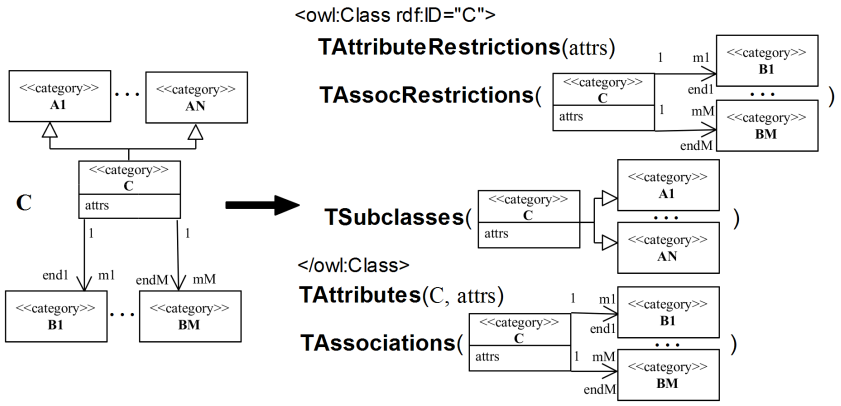

Fig. 6. Example of source-target pair defining the **TCategory** sub-transformation

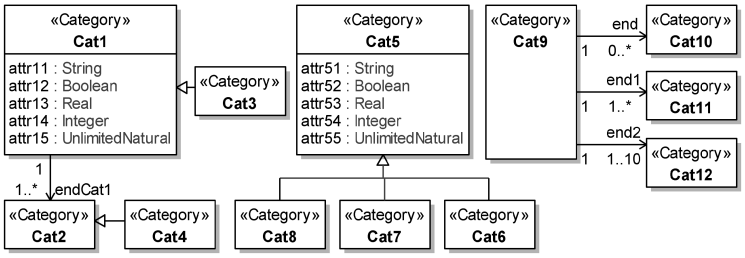

Fig. 7. Model 1 used to test U-OWL

Model	Classes	Attributes	Gen-Specs	Associations	Objects	Slots	Links
Model 1	12	10	5	4	0	0	0
Model 2	7	3	3	3	0	0	0
Model 3	2	10	0	2	4	20	4

Table 1. Simple Metrics of Test Models

To select the set of model elements to be used as input in our unit tests, we use an adequacy criteria defined as follows: at least an instance of the various templates of the M2TT design and at least an occurrence of all UML constructs used to build the source model must appear in the input models[5].

For example the model shown in Fig. 7 contains an instance of the pattern shown in the definition of **TCategory** (see Fig. 6). As we can see, class attributes are instantiated in the model with all the permitted primitive types, the multiplicities of the association ending roles (m1, mM) are instantiated in the model using different values such as 0..*, 1..* and 1..10.

4 Unit Testing of U-OWL

To perform the unit test of U-OWL we used three models, created in order to satisfy the adequacy criteria described in Sect. 3. One of these input models is shown in Fig. 7. The figures in Tab. 1 allow to grasp the size of the three models.

A summary of the results of the execution of all the unit tests (i.e. for the three models) is shown in Tab. 2. For each model we have reported the number of test cases executed on each sub-transformation and how many have failed. There are failed tests for the following sub-transformations: **TType**, **TOntology**, **TCategory**, **TAttribRestriction** and **TAssociation**. In the majority of the cases, all the tests have failed; this is due to the fact that some sub-transformations erroneously transform any possible input.

For the failed tests, by navigating the hyper link shown in the rightmost column of the HTML report (see Fig. 5) we can examine the output of the sub-transformations not satisfying the condition expressed by the tests, and thus we were facilitated to understand what the problem in their definitions is. **TType** does not produce any output when the type of an attribute is *double*. A quick inspection of the code implementing **TType** allows us to discover the error, **TType** does not transform *double*. In the same way, we are able to discover the reasons of the failure of the other tests. For instance, in the case of **TCategory**, the `owl:Class` tag is not closed in the proper way, while in the case

[5] This is a slightly different version of the criteria defined in our previous work [12].

	Model 1		Model 2		Model 3	
ST Under Test	Num. Tests	Num. failed	Num. Tests	Num. failed	Num. Tests	Num. failed
TType	10	2	5	1	30	6
TOntology	1	1	1	1	1	1
TCategory	12	12	7	7	2	2
TAttribRestriction	10	10	5	5	10	10
TAssocRestriction	4	0	3	0	2	0
TMinCardinality	4	0	3	0	2	0
TMaxCardinality	4	0	3	0	2	0
TAttribute	10	0	5	0	10	0
TAssociation	4	4	3	0	2	2
TSubClass	5	0	3	0	-	-
TInstance	-	-	-	-	5	0
TSlot	-	-	-	-	20	0
TLink	-	-	-	-	4	0

Table 2. Tests and Errors

of **TOntology** one of the name space declaration contains an URI different from the expected one. Finally, in the case of **TAssociation** the domain and range references are inverted in the generated OWL code.

Some of these errors cannot be revealed by the other kind of tests (such as the conformance and semantic test). Indeed, the error revealed by the unit test on **TOntology** cannot be revealed by any other kind of test (given that name space can be defined using any URI), meanwhile the error revealed by the unit test on **TAssociation** can be revealed only by a semantic test (given that, this kind of error produce a correct RDF/XML specification of an OWL object property, but it is not semantically compliant with the input model). All the material related (e.g. input models and HTML reports) can be found at `http://sepl.dibris.unige.it/2014-OWLTest.php`.

5 Related Work

Wimmer et al. [13] propose a technique to test model to text transformations based on tracts, that requires to transform an M2TT into a model to model one, by transforming what we have called STAs into models defined by a specific meta-model with meta-classes corresponding to folders, text files and their lines. The tests derived by tracts use the OCL extended with a "grep" function to define their oracles. The intent of this approach is quite similar to our since it consists in checking that the text produced by the M2TT has the required form. Moreover, tracts defined for sub-transformations may be used to build unit tests as those considered by our approach.

García-Domínguez et al. [7] present an approach for testing "model management tasks" within the Epsilon platform based on the unit testing framework EUnit; differently from our proposal, [7] does not consider model to text transformations, and, despite the name, does not seem to introduce any kind of "unit testing" not even for model to model transformations.

At the best of our knowledge there are no other works which deal specifically with M2TT testing or unit level model transformation testing (except our previous work [11]).

Esther Guerra in her work [8] considers model to model transformations and starting from a formal specification written using a custom specification language can derive oracle functions and generate a set of input test models that can be used to test the model transformation written using transML [9], a family of modelling languages proposed by the same author and others.

6 Conclusion

In this paper, we have presented an approach for the unit testing of model to text transformations, and up to our knowledge no other ones have been already proposed. The only requirement of our approach is that the transformation must be provided of: (1) a functional decomposition diagram showing the various sub-transformations composing it, and (2) a design specification for each sub-transformation reporting information on the textual artifacts it produces. Moreover, we have also presented a case study concerning the unit testing of a transformation from UML models of ontologies into OWL code performed following the proposed method.

As future work we plan to validate the power of the proposed unit testing approach for model to text transformations, together with the other kind of tests introduced in previous works, such as semantic and conformance (see [12]), by means of empirical experiments, for example based on fault-injection, and by applying them to other case studies concerning transformations larger and more complex than U-OWL.

References

1. Acceleo. `http://www.eclipse.org/acceleo/`.
2. Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/`.
3. Eclipse Modeling Project. `http://www.eclipse.org/modeling/`.
4. Eclipse platform. `http://www.eclipse.org/`.
5. OMG MOF model to text transformation language (MOFM2T). `http://www.omg.org/spec/MOFM2T/1.0/`.
6. B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6):139–143, 2010.
7. A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo. EUnit: A unit testing framework for model management tasks. In *Proceedings of 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS 2011, pages 395–409. Springer, 2011.
8. E. Guerra. Specification-driven test generation for model transformations. In Z. Hu and J. Lara, editors, *Theory and Practice of Model Transformations*, volume 7307 of *LNCS*, pages 40–55. Springer Berlin Heidelberg, 2012.
9. E. Guerra, J. Lara, D. Kolovos, R. Paige, and O. Santos. Engineering model transformations with transml. *Software & Systems Modeling*, 12(3):555–577, 2013.
10. A. Tiso. *MeDMoT: a Method for Developing Model to Text Transformations*. PhD thesis, University of Genova, 2014.
11. A. Tiso, G. Reggio, and M. Leotta. Early experiences on model transformation testing. In *Proceedings of 1st Workshop on the Analysis of Model Transformations (AMT 2012)*, pages 15–20. ACM, 2012.
12. A. Tiso, G. Reggio, and M. Leotta. A method for testing model to text transformations. In *Proceedings of 2nd Workshop on the Analysis of Model Transformations (AMT 2013)*, volume 1077. CEUR Workshop Proceedings, 2013.
13. M. Wimmer and L. Burgueño. Testing M2T/T2M transformations. In A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. J. Clarke, editors, *Proceedings of 16th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2013)*, volume 8107 of *LNCS*, pages 203–219. Springer, 2013.