A Method for Testing Model to Text Transformations

Alessandro Tiso, Gianna Reggio, Maurizio Leotta

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

Abstract. Model Transformations are the core of Model Driven Development; hence for generating high quality artifacts we need methods able to assure the quality of model transformations. In this work, we focus on Model to Text Transformations and propose a method, composed by a set of integrated approaches and a series of guidelines, for testing model transformations. We classify the test approaches composing our method relying on the intent on which they are carried out. Moreover, to select input models used to build test cases, we propose a definition of adequacy criteria and coverage.

1 Introduction

In this work, we present a method for testing Model to Text Transformations (MTTs), which uses a set of integrated approaches. It is part of our **Me**thod for **D**eveloping **M**odel **T**ransformations (*MeDMot*), but most of the underlying ideas and techniques can be applied to test any MTT. *MeDMot* aims to support the development of transformations from UML model to text, precisely it considers MTTs of the kind shown in Fig. 1. It prescribes how to: define the requirements of a transformation, then how to design, implement and test it. Thus, the testing activity can take advantage of the specificity of requirements and design prescribed by *MeDMot*. Moreover, the model transformation architecture and the model transformation languages (ATL and Acceleo) used are the same specified in our previous work [8].

Model transformation testing is a complex task [2], since the complexity of the inputs (e.g., complete UML models instead of numerical values), and consequently a large effort is required to produce them, 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 for model transformations is more difficult than selecting input for programs testing because they are more difficult to be defined in an effective way (e.g., compare defining a set of input values for a program with defining a whole UML model). Hence, also defining adequacy criteria for MTTs is again more difficult than in the case of programs. For these reasons, we try to define a method for testing MTTs, whose application is simple and that can scale with the size of the tested model transformations. Moreover, our approach to MTTs testing is pragmatic and has its main purpose in providing guidelines to design and implement tests for them taking into account the extreme variability of their targets.

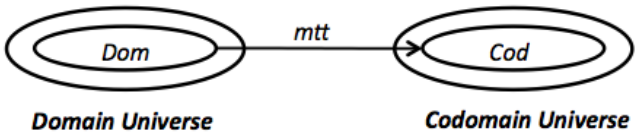

Fig. 1. A generic MTT *mtt*

Our work has been motivated by the need to test various MTTs that we developed by applying *MeDMot*. All of them have as source a class of UML models built using a specific profile, whereas the targets are respectively: (1) Java Desktop applications (U-Java); (2) SQL scripts (U-SQL); (3) OWL Ontologies (U-OWL); (4) Coloured Petri nets suitable for the CPN tools[1] (U-Petrinet).

In Sect. 2 we provide a classification for MTT testing, Sect. 3 presents the adequacy criteria for MTTs testing, Sect. 4 proposes an approach to MTT unit testing, and Sect. 5 gives some hints on how the testing activity can be automated. Finally, Sect. 6 outlines the guidelines of our method for testing MTTs. Related work and conclusion are in Sect. 7 and 8 respectively.

1.1 Preliminaries

A *structured textual artifact* (STA) is a set of text files, written using one or more concrete syntaxes, disposed in a well-defined structure (positions of the files in the file system).

In this paper we consider the kind of MTTs shown in Fig. 1. The *Domain Universe* is the set of the UML models built using a specific profile, *Dom* is a subset of *Domain Universe* containing all models assumed to be a correct input for the transformation. The *Codomain Universe* is a set of STAs, *Cod* is the subset of *Codomain Universe* containing all the STAs assumed to be a correct result of the transformation. *mtt* is a function that transforms elements of *Dom* into elements of *Cod*.

Both the elements in the domain and in the codomain have associated a semantics (e.g., U-SQL: UML models representing persistent data structure → SQL statements, U-Java: UML models representing desktop applications → Java programs), which the transformation developer should know. The transformation requirements should be expressed in terms of the semantics of the transformation source and target, e.g., in the case of U-Java, every persistent class should be transformed into a persistent Java class of the application generated or, in the case of U-SQL, every class stereotyped by ≪table≫ should be transformed into a SQL statement for table creation.

Instead, the design of a transformation should define how domain elements are transformed into fragments of the STAs. In the case of U-SQL the design prescribes that a class stereotyped by ≪table≫ and named A is transformed into the SQL statement like CREATE TABLE TABLE-A.

U-OWL: the running example. U-OWL is an MTT from UML models of ontologies to OWL.

[1] `http://cpntools.org/`

An ontology model is composed by a class diagram (the StaticView) defining the structure of the ontology in terms of categories, specializations, and associations, and possibly by an object diagram (the InstancesView) describing the information about the individuals of the ontology (i.e., which are the individuals, the values of their attributes and the links among them). The object diagram must be defined with respect to the class diagram, i.e., its instances must be typed by classes present in the latter, and similarly its links and slots must correspond respectively to associations and attributes in the latter. The used UML profile is composed of two stereotypes: ≪category≫ and ≪instances≫. A UML class stereotyped by ≪category≫ represents a class definition of the ontology, its individuals will be defined in the instances view. A UML class stereotyped by ≪instances≫ must have a list of literals and will define simultaneously a class definition and its individuals, defined by its literals. Such UML class cannot have attributes. Moreover, there is a set of well-formedness rules characterizing *Dom* that, for space reasons, we do not show here, such as: "all the attributes of a UML class must have multiplicity equal to one and be typed by a UML primitive type". The output of the transformation is a text file describing the ontology using the RDF/XML for OWL concrete syntax. The transformation must produce a OWL definition of an ontology having exactly the classes and the instances described by the input model together with the features (e.g., attributes, relationships, slots) defined again by the model.

2 Transformation Test Classification

A transformation *test case* is a triple $(IM, check, expct)$, where $IM \in Dom$ is an input model, $check$: $Cod \rightarrow CheckResult$ is a total function and $expct \in CheckResult$. *check* represents some observations on the output models, where the result of such observations will be some elements in *CheckResult* (obviously, *CheckResult* is the set of the observations on the output models), and *expct* is the expected result of the observations on the transformation of *IM* (i.e., what forecasted by the oracle). The test is passed if $check(mtt(IM)) = expct$. A *test suite* is a set of test cases.

We classify the MTT tests on the basis of the *intent* with which they are carried out (taking also inspiration from [1]).

Conformance tests are made with the intent of verifying if the MTT targets belong to the *Cod*. In general this means checking that the textual artifacts produced by the transformation have the required structure (e.g., a folder containing at least two files and no subfolders) and that the various files have the correct form (e.g., they are correct with respect to a given BNF or XML schema).

Referring to Fig. 1, conformance tests are done with the intent of verifying that all the elements belonging to the *Dom* are transformed by *mtt* into elements of the *Cod*. The *check* function must verify some properties characterizing the codomain. For instance, in the case of a transformation producing Java programs, a check may verify that they are syntactically correct/they use only standard API/all their identifiers are lower case; whereas for an MTT producing HTML documents a test may check that they are correctly visualized by a

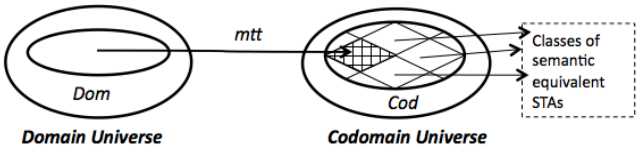

Fig. 2. Transformation function – Semantic test

specific browser[2] or that they describe a web site respecting some accessibility requirements.

If a conformance test fails, then we firstly have to look at the constraints of the domain and the codomain definition to see if they are too weak/too strong, and later we have to examine the *mtt* design and implementation. To better clarify this point consider the following example: let *mtt* a transformation from UML models representing databases into SQL, if the constraints over the transformation domain do not prohibit class names with whitespaces, then a substantially correct transformation will transform these element names into wrong table identifiers, and conformance tests will fail. In this case, the correct developer action is to refine the definition of the domain, so to forbid UML element names containing whitespaces.

In a conformance test case (*IM*, *check*, *expct*), *check* and *expct* do not depend on *IM*, because these two test case ingredients are built only by looking to the definition of *Cod* and the transformation requirements.

Referring to the U-OWL case, an error discovered by conformance tests is the following: the transformation forgot to add the closing tag to the definition of the OWL class corresponding to a UML class. Trying to open with Protege[3] the OWL file produced by the transformation of whatever trivial model (i.e., containing at least a UML class) we are informed of an exception occurred during the parsing of the input file.

Semantic tests are made with the *intent* of verifying that the target of the MTT has the expected semantics. Referring to the Fig. 2, semantic tests are done with the intent of verifying that elements belonging to the *Dom* are transformed into elements of the *Cod* belonging to the right class of equivalent STAs (class of equivalent STAs are depicted with diamonds in the picture). The *check* function using to build the test cases has to verify semantic properties typical of the codomain nature, using methods and techniques again typically of that context (e.g., Java programs, OWL ontologies)

To clarify the concept we give two examples: (1) if the transformation target are Java programs, then a semantic test may verify if some class operation has the required behaviour by means of classical tests or that an interactive program may execute some specific scenario; (2) in the OWL case the meaning of a OWL file is an ontology made of semantic categories (OWL classes), containing individuals, and of inclusion/sub-typing relationships between categories, thus the semantic tests may check if the represented ontology has all the OWL classes described

[2] obviously this test will be performed by a human being

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

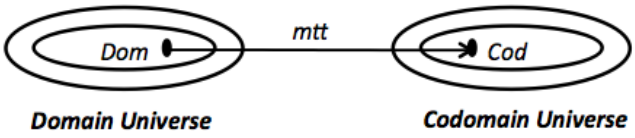

Fig. 3. Transformation function – Textual test

in the UML input model, or that an OWL class has all the required individuals, or that an OWL class is a sub-class of another one.

Semantic tests should be built taking into consideration all the techniques and methods already existent for the nature of the *Cod*. For instance, if the *Cod* is the set of all the Java projects for desktop applications, then we can employ the techniques used to test (the semantic of) a Java program. Referring to the U-OWL case, an error discovered by semantic tests is the following: the input model is composed by a class diagram containing a hierarchy of classes each one stereotyped with ≪category≫ and an object diagram containing only one object instance of one of the leaf classes. The output produced must contain an ontology in which are described the same class hierarchy and an individual belonging to a leaf class. A semantic test could be the "realization of an individual"[4] reasoning task that may be performed by a reasoner like FaCT++ [9].

Semantic tests depend on the source model. Indeed, the form of the check function depends on what elements are contained in the source model. In some cases the expected results can be automatically derived from the source model (see Sect. 5). Semantic tests are very important because allow to find the most serious failures, for instance when the transformation requirements are not met, even if the transformation result is a running Java program or an OWL ontology correctly shown by Protege.

Textual tests are made with the *intent* of verifying that the textual elements comprising the STA target of the MTT have the required form. Referring to the Fig. 3, textual tests are done with the intent of verifying that each element belonging to the *Dom* is transformed into the right element of the *Cod*. The *check* function must verify that the result of the transformation considered as a pure textual artifact has the right form. Textual tests depend on the input models, similarly to the semantic tests.

For example: (1) if the *Cod* is JAVA code, then a textual test may verify if there is a correct number of files, in the right position in the file system and with the right names; (2) if the *Cod* is an ontology written using OWL, then a textual test may verify if it contains the right number of class structure definitions. Referring to the U-SQL case, an error discovered by textual tests is the following: the target contain the string "AT char(60)" in correspondence of an attribute *AT* typed by the UML basic *string* type in the source model, instead of the string "AT char(64)" (as required by the design). In this case the string found in the target is correct with respect to a given BNF, but it is not the

[4] that is, find all classes which the individual belongs to, especially the most specific one

expected one (in our example the string "AT char(64)"). We want to emphasize that this kind of errors are not revealed by the conformance test.

3 Transformation Test Adequacy Criteria

A *test adequacy criterion* is a criterion that drives the selection of *characteristics* for input models that will be used to generate the test models needed to build the test suites (adapted from the definition found in [2]). We use model characteristics instead of whole models for criteria definition, because defining whole models requires a substantial effort and to fix a large number of details not relevant (e.g., to require to have a UML model with a class with a given stereotype and 3 operations and no attributes versus to have to produce a complete model defining also the names, the parameters, and the result types of such operations).

The *coverage* is the ratio between the number of characteristics identified by a test adequacy criterion that can be found in the input models of a test suite and the total number of characteristics identified by the same test adequacy criterion.

In the following we describe three adequacy criteria. Let S be the set of the stereotypes and tagged values comprising the UML profile used for defining Dom; let CD be the set of the UML constructs/diagrams allowed to appear in the models in Dom considered relevant by the transformation developer; and let $M = S \cup CD$.

Referring our running example U-OWL, S={≪category≫, ≪instances≫ } and CD={class, association, object, link, Class Diagram, Object Diagram}. In this case the developer did not considered relevant to insert in CD also the attributes and the slots.

Criterion 1. The predicates on Dom defining this criterion are: $pred_x : \mathsf{Dom} \rightarrow Bool$ defined by $pred_x(Mod) \Leftrightarrow x$ appears in Mod, for $x \in M$, $Mod \in \mathsf{Dom}$; thus we have the same number of predicates and elements of M. Each test model must contain at least one of the elements of M. Starting from this criterion we can define other criteria, simply requiring that each predicate considers more than one element of M, e.g., if $x, y \in M$ and $x \neq y$, then we can define a predicate $pred_{x,y} : \mathsf{Dom} \rightarrow Bool$ such that $pred_{x,y}(Mod) \Leftrightarrow x, y$ both appear in Mod. So, for example in our case the set M is composed by eight elements thus, we need, at least, to define 8×7 predicates.

Criterion 2. This adequacy criterion takes into account the features of the elements in M, and is defined by the set of predicates that evaluate the presence in the input models of a set of relevant combinations of such features. The developer must consider that some combination of them may be not allowed by the definition of Dom. Each test model must contain at least one of these sets of relevant combination of features. In the U-OWL case, the features of class (that belongs to M) allowed in the Cod (just the attribute) will contribute to the criteria, for instance, with predicates checking the presence of a class without attributes and of a class with 3 attributes.

Criterion 3. The adequacy criterion takes advantage on the way we give the MTT design. The design of each transformation function is given by means of

relevant source-target pairs. Each pair is composed by a left side, that shows a template, and a right side that shows the result of the application of this function on model fragments obtained instantiating such template. So, criterion 3 is defined by the set of predicates checking that the various templates are instantiated on the input models. Each test model must contain at least one of the templates showed in the left side of the source-target pairs. We cannot show the application of this criterion in the U-OWL case, because we do not report its design.

4 Transformation Unit Test

A non-trivial MTT will be built by composing many sub-transformations, that transform different - but potentially overlapping - parts of the input models; for example in the case U-OWL we have a transformation of classes (contained in the StaticView), and a transformation of objects (contained in the InstancesView); thus we can consider as subject of testing the whole transformation or its sub-transformations. The various sub-transformations may be arranged in a kind of call-tree (we use a diagram similar to the structure chart, that we call *decomposition diagram*), where the nodes are labelled by the sub-transformations themselves and the children of a node labelled by S are the trees corresponding to those called by S. A sample of the structure of a MTT in terms of sub-transformations is shown in Fig. 4.

Thus, we can decompose the whole transformation in parts, each one composed by the sub-transformations belonging to one of the subtrees, and test these parts separately. For example in Fig. 4 we can test separately the parts of the transformations corresponding to the subtrees *T1* and *T2*. The testing of the transformation parts will use test cases built with model fragments, but it should be checked that still they satisfy the restrictions imposed by the definition of *Dom*, i.e., they may be extended to become models in *Dom*. The developer should also choose a decomposition that allows to have transformation parts that generate self-consistent structured textual artifacts, to allow the semantic testing. Using this approach we can build a kind of unit testing of a MTT. Indeed, each part of the transformation can be tested separately.

5 Transformation Testing Automation

We consider two levels of testing automation: (1) automate the generation of the artifacts needed for testing and (2) automate the execution of testing. In the former case the check function and sometimes also the expected results are automatically generated from the elements contained in the source model. In the

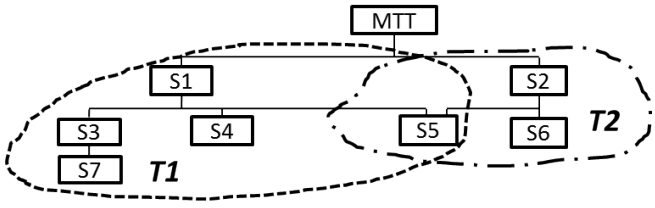

Fig. 4. A sample of MTT structure

latter case the execution of testing and sometimes also the retrieval of testing results can be partially, or totally automatized.

Automate test generation. If the generation of testing artifacts ***can*** be described in an algorithmic way, the needed artifacts can be produced during the execution of the MTT to be more precise by an extension of MTT that together with the target produces also the test cases components, without adding any information to the source models. This is possible when the structure of the artifacts and the input data for the test cases depend only on the content of the source models. For example, if we consider the U-Java case, we have that the source models may contain persistent classes (those stereotyped by ≪store≫), then the generated persistent entities may be tested generating a JUnit test that creates number of entity instances and then retrieve them.

If the generation of the needed testing artifacts ***cannot*** be described in an algorithmic way, we can try to add some parts in the source model and extend the transformation to map them into tests. For example in the case of U-Java, we do that by inserting in the source model test classes and test operations, that drive the generation of executable JUnit tests in the target code.

Automate test execution. Automating the execution of testing becomes an important issue when the number of test cases grows, and should be supported by tools. Depending on the nature of the codomain of the transformation, tools designed expressly for automate the execution of testing may already exist (e.g., JUnit for Java). After that the testing artifacts are generated (automatically or by the transformation developer), a specifically designed application can execute the test suites and, in some cases, can generate reports containing the result of testing. For example, if we consider U-Java, the testing execution can be automated by means of Maven [7]. All the test generated artifacts consist, in this case, of Java code and configuration files placed both in the same project in which the application code is generated, but in a separate folder, that Maven recognizes as the tests container. The developer only needs to activate the build process. During the build process tests are executed and report files are produced.

6 Transformation Testing Method

In this section we outline the guidelines of our method for testing MTTs.

We believe that there is a "natural" order with which tests should be performed: starting from tests which have the intent of discovering coarse errors to tests which want to discover finer errors. Thus, our testing method requires to first execute the conformance tests, then the textual tests and finally the semantic tests. Indeed, conformance tests are those that reveal the largest number of errors, because they are able to detect all MTT implementation errors related to generate text not following the required concrete syntax, e.g., forgetting to close a parenthesis or forgetting a space between two keywords (which in our experience are very common). Instead, semantic tests are able to detect more serious errors due to the requirements understanding or implementation. Finally, textual tests are able to discover if the semantic of the MTT output is obtained using the wanted syntactic constructs.

To build each test suite we have to build a set of test cases consisting of an input model, a check function and the expected result of the check (see Sect. 1.1). For each kind of tests, we fix the form of the checks, and build three test suites each one using input models selected using the three adequacy criteria defined in Sect. 3. Thus, for each kind of tests (those introduced in Sect. 2) we build three suites. Hence, we have a total of nine test suites. The test suites should be executed in the following order: first the three test suites built with the intent of detecting conformance errors, then the three test suites built with the intent of detecting syntactic errors and finally those having intent of discovering semantic errors. Within each group, the test suites built using the *Criterion 1* should be executed first, followed by those built using the *Criterion 2* and finally those built using the *Criterion 3*. The following table summarises the tests suites that must be built and the order of execution.

Test Type	Criterion 1	Criterion 2	Criterion 3
conformance	1	2	3
textual	4	5	6
semantic	7	8	9

If there are constraints to the time available to testing, we suggest to prioritise the building and execution of the test suites made using criterion 1, then those made with criterion 2 and finally using criterion 3. Moreover, we suggest to privilege conformance tests, then textual and finally semantic tests.

Each input model may be instance of more than one template, but as guideline we suggest to build small input models that are instances of the minimum number of templates.

7 Related Work

At the best of our knowledge there are no other works which deal specifically with MTT transformation testing (except our previous work [8]).

Fleurey et al. in their work [3] define test adequacy criteria based on input metamodel coverage adapting UML-based test criteria defined in [6]. Moreover, they report that this kind of adequacy criteria select as input for testing the model transformation, a significant amount of data irrelevant w.r.t. the model transformation and so, they describe a way to define a sub-set of the input metamodel selecting metamodel elements that are relevant for the model transformation. Our definition of adequacy criteria enable us to select only input models relevant for our MTT under test. Moreover, we define the various criteria using the concrete syntax of the input models and not the abstract syntax (i.e., the metamodel).

Esther Guerra in her work [4] 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 [5], a family of modelling languages proposed by the same author and others. In our case input models are not generated, but the adequacy criteria we propose assure that

they cover a good portion of the interesting properties of the transformation. Moreover, this work considers only Model to Model transformation, instead we are interested more in Model to Text Transformations.

Amrani et al. in their work [1] report on the building of a catalog of model transformation intents and the properties model transformation has to have. They describe a schema for the model transformation intent catalog where each intent has a set of attributes and properties that must be satisfied. Moreover, they catalogue a list of common transformation intents and model transformation properties among that we can find the *type correctness* and *property preservation*, that in some way we use in our classification of tests. Finally, we take inspiration from this work in the idea of the classification of testing approaches.

8 Conclusion

In this paper, we have proposed a method for testing Model to Text Transformations, which uses a set of integrated approaches. It is part of our method for developing model transformations (*MeDMot*), but most of the underlying ideas (and techniques) can be applied to any MTT context. Moreover, we have provided a novel definitions of *adequacy criteria* and *coverage* specific to the MTT context. As future work we plan to conduct an empirical investigation using mutation techniques to see what kind of errors are really revealed by the three proposed types of testing.

References

1. M. Amrani, J. Dingel, L. Lambers, L. Lúcio, R. Salay, G. Selim, E. Syriani, and M. Wimmer. Towards a model transformation intent catalog. In *Proc. of AMT 2012*, pages 3–8. ACM, 2012.
2. B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, 2010.
3. F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: testing model transformations. In *Proc. of MODEVA 2004*, pages 29–40, 2004.
4. 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.
5. E. Guerra, J. Lara, D. Kolovos, R. Paige, and O. Santos. Engineering model transformations with transml. *Software & Systems Modeling*, 12(3):555–577, 2013.
6. J. A. Mc Quillan and J. F. Power. A Survey of UML-Based Coverage Criteria for Software Testing. Technical report, Department of Computer Science, Maynooth, Co. Kildare, Ireland, 2005.
7. F. P. Miller, A. F. Vandome, and J. McBrewster. *Apache Maven.* Alpha Press, 2010.
8. A. Tiso, G. Reggio, and M. Leotta. Early experiences on model transformation testing. In *Proceedings of the 1st Workshop on the Analysis of Model Transformations*, AMT 2012, pages 15–20, New York, NY, USA, 2012. ACM.
9. D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: system description. In *Proc. of IJCAR 2006*, pages 292–297. Springer, 2006.