Model transformation testing: oracle issue

Jean-Marie Mottu[1], Benoit Baudry[1], Yves Le Traon[2]

[1]*IRISA,*
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France
{jmottu, bbaudry}@irisa.fr

[2]*TELECOM Bretagne,*
2 rue de la Châtaigneraie CS 17607,
35576 Cesson Sévigné, France
yves.letraon@telecom-bretagne.eu

Abstract

The definition of an oracle function for model transformation is challenging because of the very complex nature of models resulting from a transformation. Validating the correctness of an output model requires checking a large number of properties on the structure and semantics of this model. The oracle function can thus be very complex if it checks every property. In this paper, we identify and discuss important issues that must be tackled to define model transformation testing oracles. We also propose several oracle functions and analyze how they take advantage of different model driven engineering techniques.

1. Introduction

Model transformations are intensively used for model-driven development (MDD) in order to automate critical operations in the development such as refinement, code generation or refactoring. The automation of these operations should increase reuse from one project to the other and thus save time and effort. However, this automation also introduces additional risks of errors due to faulty transformations. Thus, systematic and effective testing of model transformations is necessary to the success of model-driven development.

Two problems need to be solved when tackling model transformation testing: efficient test data selection and the definition of an oracle function. An oracle function analyzes the validity of models produced by the model transformation. This paper focuses on this second problem.

The definition of an oracle function for model transformation is challenging because of the very complex nature of models resulting from a transformation. The result of a model transformation is a model that conforms to a metamodel. The metamodel defines both the structure of the model in terms of classes and relationships as well as its semantics. A model is manipulated as a graph of objects that instantiate the classes from the metamodel and that are related according to the relationships and constraints defined in the metamodel.

This paper discusses some of the challenges related to the definition of an oracle function for model transformation testing. We highlight several issues related to developing oracles depending on the testing context.

With respect to oracle definition, we will discuss the use of several model-driven engineering (MDE) techniques. We propose and analyze six possible oracle functions which check models resulting from a transformation. These functions differ on the information they require from the tester.

We propose the different solutions for the oracle and present an example that illustrates the different approaches to the oracle definition problem. Based on this first experiment, we analyze the different trade-offs that must be considered when choosing a particular oracle function for a model transformation.

In section 2, we introduce the model transformation testing issue. In section 3, we go into the details of the oracle definition that we have studied. We discuss the problems that arise when developing and applying the oracles. In section 4, we use a model transformation to illustrate the definition of such oracles and we analyze this work in the section 5.

2. Model transformations and their testing

Model-Driven Development (MDD) aims to provide automated support for creating and transforming software models. Effective support for model transformations is thus key to successful realization of MDD in practice. In this section, we introduce the model transformations and discuss the challenges that must be addressed when testing them. One example is introduced to assist the explanations.

2.1. Model transformation

Figure 1 shows a generic transformation framework that provides the context for discussion in this paper. A model transformation manipulates concepts that are specified in the source and target metamodels (which can be different). These metamodels describe the static structure of the models that are manipulated by the transformation. In the transformations we have developed, these metamodels conform to the MOF [1]. In some cases, these metamodels are augmented with constraints (expressed in OCL for example) that more precisely constrain the structure of models that are manipulated by the transformation. In the case of the UML metamodel, these constraints are the well-formedness rules.

A transformation takes an input model that conforms to the source metamodel and produces an output model, which conforms to the target metamodel. In the following, we consider transformations that take a single input model and produce a single output model.

The precondition shown in Figure 1 further constrains (in addition to the source metamodel and its associated constraints) the type of models that can be input to the transformation. The post condition specifies expected properties on the output model as well as properties that link the input and the output models. These additional constraints are of the same nature as the constraints on the metamodels, but they are specific to the transformation.

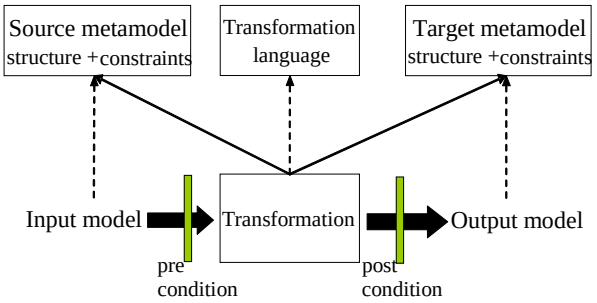

Figure 1. General framework for model transformation T

2.2. One significant model transformation

We illustrate this paper with a transformation which transforms a class model to an RDBMS model as an illustrative example. Its specification has been proposed in the MTIP workshop [2]. It is made of a set of complex rules. The implementation of such a system requires complex operations on the input model with recursivity, navigations with transitive closure, and several passes.

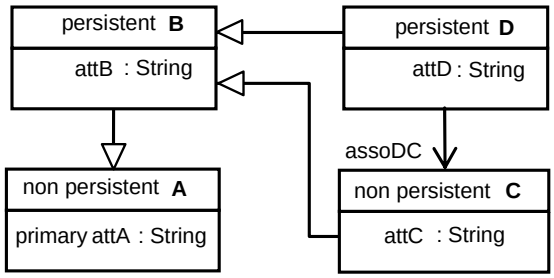

Figure 2. Input model Mt1

For instance, the class model of the Figure 2 is made with several classes, several rules specify how the persistent classes, their attributes and their associations should be transformed into tables, columns and keys. Following the specification, this sample input model is transformed into the RDBMS model of the Figure 3. Any output model conforms to the RDBMS metamodel illustrated in Figure 4.

The structure described in this metamodel can be reinforced with constraints. For instance, an RDBMS model cannot contain two tables with the same name. This constraint written in OCL is:

```
context RDBMSModel
inv:
self.table.forAll(t1,t2|
   t1.name = t2.name implies t1 = t2
                  )
```

We do not detail all the specification of the transformation T but only a set of rules that we composed in a homogeneous rule:

```
Ru: "The persistent classes, and only
these ones, are transformed into tables
with same names, except if they inherit
directly or not from another persistent
class"
```

In addition to the different rules, the specification also restrains the input domain with contracts. For instance, one can impose that in any class model, each class should have at least one primary attribute. The underline reason is that the corresponding table will have at least one column which is its primary key. This constraint written in OCL is:

```
context
class2rdbms(in:ClassModel):RDBMSModel
pre :
in.classifier
       .select(c|c.oclIsType(Class))
       .forAll(cs|
cs.attrs.exists(a|a.is_primary = true)
              )
```

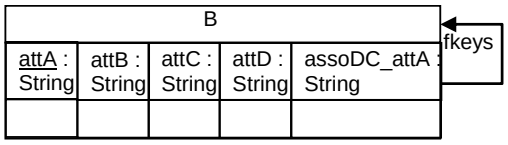

Figure 3. Output model from T(Mt1)

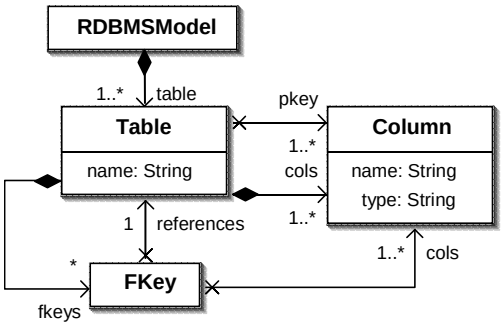

Figure 4. Output RDBMS metamodel

2.3. Model transformation testing

Model transformations manipulate models, which are very complex data structure. This makes the problems of test data generation and selection, as well as the oracle definition very difficult.

For example, to test the class to RDBMS transformation, test data generation consists in building class models such as the one displayed Figure 2. We can imagine that the test set should contain one model with inheritance, one with no inheritance, one with one persistent class, one with more than one persistent class… So the first challenge is to define criteria for test data generation. Then, there is still a lot effort required to build all the models necessary to satisfy the criteria.

Then, the oracle function has to validate that the produced table is correct, according to the requirements and the class model provided an input to the transformation. For example, it must check that if there is only one persistent class A in the input model, then there is only one table, called A, in the output table. Thus, this oracle has to manipulate two models and check a number of properties. The challenge here is to formalize all the properties and to express them with respect to input and output modeling languages.

The issues related to input test data generation are outside the scope of this paper. Initial results are proposed in other works. Fleurey et al. [3] define several test adequacy criteria, adapting category-partitions on the input metamodel of the transformation. Automatic generation of models has been studied by Ehrig et al. [4], and in Sen et al. [5], we also propose an approach for test models generation.

In this paper, we focus on the issue of defining oracle function, assuming that a set of test data can be provided. As we detail it in the following, there are many different ways to define this function, depending on the effort provided and on the amount of information that is available (formal specification, expected output, etc.). We believe that the analysis of the different oracle functions will lead to a better understanding of the challenges for model transformation and will allow us to compare the different solutions in the future for a better and systematic engineering of test generation.

3. Oracle issue

This section introduces the oracle issue in model transformation testing.

3.1. Validate the output models

The oracle checks the validity of the output model returned by the transformation of one test model. It analyzes a model and returns the verdict for the test case.

Few works mention the oracle function for model transformation, and they usually consider that the expected model for a particular run of the transformation is available [6, 7]. Thus, they transform the problem of oracle definition into a problem of model comparison. Although this approach has to be considered and efficient solutions for model comparison will help the definition of an oracle function, we believe that considering the oracle only through this perspective is too restrictive. First, the expected model is not easy to obtain, and the tester might face difficulties to produce expected models for all the test cases. Secondly, there are several other ways to analyze the output model and produce a verdict that should not be neglected because they could fit more easily the tester needs. Finally, we believe that a model transformation testing oracle should not be reduced to a data but has to be considered as a full function with its parameters.

In order to analyze the definition of an oracle in a broader way than simple model comparison, we consider the oracle as a parameterized function. The first parameter is the output model returned by the transformation. The second parameter must be provided by the tester, and we call it the "*oracle data*". This data provides details to verify the output model. It is the main parameter of the oracle. For instance, it can be the expected model of the test case; it can also be

the test model if an oracle needs to extract information from it to check the output model.

In the following, we analyze the different data that can be provided and the different functions that can be defined, depending on the oracle data.

3.2. Three MDE techniques to implement oracle functions

In this section we introduce three Model Driven Engineering (MDE) techniques that manipulate and analyze models and that can be used to implement model transformation oracles.

3.2.1. Model comparison

Current MDE technologies and model repositories store and manipulate models as graphs of objects. The complexity of these data structures makes it difficult to provide an efficient and reliable tool for comparison. In the general case, model comparison is equivalent to computing graph isomorphism problem which is NP-complete. However, several studies have proposed simplified versions of this comparison that can be used at a much lower computation cost. Porres et al. [8] present a theoretical framework for performing model differencing. However, they rely on the use of unique element identifiers for the model elements. This assumption cannot hold when the models are produced by different means. Other algorithms based on the metamodels despite the objects identifiers have been proposed. In [6], Lin et al. proposed such an algorithm. Little studied, the model comparison gets to be implemented in tools like EMFCompare [9].

The model comparison can be used in different oracle functions. They compare a reference model with a model resulting from the transformation of the test model. The reference models can be available or can be obtained. Hence they can be compared by the oracle.

3.2.2. Contracts

Pre and post conditions form the contract for a method. They are assertions that are evaluated before and after the execution of one method [10]. Contracts can be defined for a model transformation. The pre condition constrains the set of licit models and the post condition declares a set of properties that can be expected on the output model.

Several researchers have studied the use of contracts as a partial oracle functions in object oriented system [10, 11]. This approach can be adapted to define an oracle for model transformation. In previous work [12], we proposed a process for specifying and implementing model transformations oracle with contracts expressed in OCL. However, contracts can also be implemented with other languages and tools, like Kermeta or a rule-based transformation language like ATOM3 [13]. In [14], Kolosov et al. present another way to link the output and the input models with rules. They are based on the comparison of the objects of the input and output models, and they define contracts in such a manner between input and output models. In [15], Küster et al. have also noticed that constraints can be used as oracle.

3.2.3. Pattern matching

A pattern can be defined as a “piece of model” or a set of model elements. Pattern matching then consists in checking the presence of a pattern in a model. We present two techniques to write the patterns: with assertions, or with model snippets.

In this paper, we consider patterns expressed as OCL assertions or as model snippets [16]. A snippet is a “piece of model” where every object is an instance of a metaclass defined in the metamodel, and the model snippet is a subset of a model that conforms to the metamodel. Samples are explained in the illustrative section 4.1.5. Having patterns expressed in this way, it is possible for the tester to write them using the same editor that he uses to write test models.

For the oracle, the patterns express constraints on the output model. In that sense, they can be considered as post-conditions, but contrary to the contracts, the patterns focus on a specific output model. Thus, patterns can be considered as assertions that should be true when running the transformation with a particular test data. Each assertion or a conjunction of several ones can be associated to a test case as the oracle data of an oracle function. It should be true to ensure the validity of the corresponding output model and the success of the test.

3.3. Six model transformation testing oracles

Six solutions are thus available to obtain the oracle when executing a test data on a model transformation.

1- Oracle using a *reference model transformation*:

A comparison is made between the output model (`mtout`) returned by the transformation of the test model and a reference model returned by the reference model transformation.

The tester should provide an oracle data which is the reference version (`R`) of the model transformation. This reference transformation can produce the reference model from the test model (`mt`).

The oracle function is the function O_1 as such:

```
O1(mtout , (R, mt)) : Boolean is
do
     result := compare(mtout , R(mt))
end
```

2- Oracle using an *inverse transformation*

A comparison is made between the test model `mt` and the model obtained after two transformations of the test model: the first with the transformation under test and the second with the inverse transformation.

The tester should provide this *inverse transformation* (`I`) as oracle data. But it is only possible if the model transformation is an injective function (which is unlikely), otherwise the transformation can not be undone.

The oracle function is the function O_2 as such:

```
O2(mtout, (I, mt)) : Boolean is
do
     result := compare(mt , I(mtout))
end
```

3- Oracle using an *expected output model*

A comparison is made between the output model (`mtout`) returned by the transformation of the test model and an *expected model* (`mtexpected`) provided by the tester.

The oracle function is the function O_3 as such:

```
O3(mtout, mtexpected) : Boolean is
do
     result := compare(mtout, mtexpected)
end
```

4- Oracle using a *generic contract*

A generic contract in an oracle function is a post condition of the transformation which constrains the outputs depending on the inputs.

The oracle checks if the output model satisfies the generic contract depending on its corresponding test model (`mt`). The tester should provide a generic contract (`Cg`) which is able to analyze both the test model (`mt`) and its corresponding output model (`mtout`). This contract can check the validity of this output model according to a part more or less important of the specification.

The oracle function is the function O_4 as such:

```
O4(mtout, (Cg, mt)) : Boolean is
do
     result := (mtout,mt).satisfies(Cg)
end
```

5- Oracle using an *OCL assertion*

The oracle checks if the output model satisfies the OCL assertion.

The tester should provide an OCL assertion (`Cd`) which is able to analyze the output model. This constraint doesn't consider the test model, it is then dedicated to a test case and its test model, it only checks the validity of their corresponding output model. The tester will express in this constraint the properties the output model has to contain. It is not mandatory to express all the properties because this task can be more simply made with an *expected model*.

The oracle function is the function O_5 as such:

```
O5(mtout,Cd) : Boolean is
do
     result := mtout.satisfies(Cd))
end
```

6- oracle using *model snippets*

The oracle checks if the output model (`mtout`) contains `n` model snippets (`ms`).

The tester should provide a pattern in the form of a list of model snippets, each one is associated to a cardinality and a logical operator. These two last express how many times the model snippet should be found in the output model (`mtout`).

The oracle function is the function O_6 as such:

```
O6(mtout,list{(ms,n,op)}) : Boolean is
do
     result := list.forAll(
     compare(nb_match (mtout,ms), n, op))
     //compares 2 numbers depending on
     //a logical operator op,
     //and returns a boolean
end
```

4. Illustration

In this section, we illustrate the implementation of an oracle with the six different oracle functions we propose.

4.1. Implementation of the test cases

First the tester gets a set of test cases. We consider only the test model of the Figure 2. With the initial specification, the model is transformed into the RDBMS model of the Figure 3, only a table B is created.

To illustrate the reusability of the different oracles, we create an evolution T' of the transformation T and the rule Ru becomes:

```
Ru': "The persistent classes, and only
these ones, are transformed into tables
with same names"
```

With this new specification, the input model is transformed in the RDBMS model of the Figure 5. The persistent classes with persistent parent are also transformed and the output model contains an additional table D.

In the following, we consider the definition of oracle functions to check the validity of these output models considering the rules `Ru` and `Ru'`.

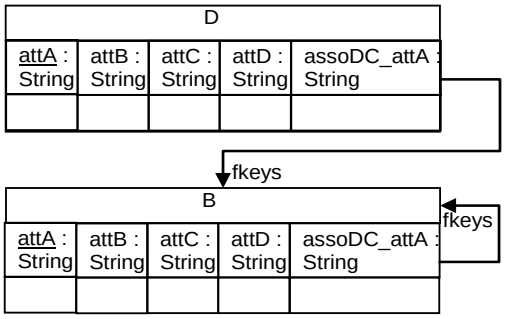

Figure 5. Output model from T'(Mt1)

4.1.1. Reference model transformation

The implementation made with Kermeta [17] is a program with 113 lines of code, in 11 operations. The tester could use an other implementation of the model transformation under test, such as one made for this workshop by Lawley and Steel [18]. It's a functional implementation written in Tefkat, made with 94 lines of code, in 8 patterns and 5 rules. The complexity of this second implementation is important, its validity is no more confident, and moreover when the tester will have to adapt it to the new specification, he will need to learn a new language, understand the implementation and modify it correctly. Hence, even if the tester can take advantage of an existing reference model transformation, its difficult reuse with a new version and the risk inherent at its complexity don't ensure the quality of this oracle. The use of a reference version is too much complex, its writing is a developer task and not a tester task.

4.1.2. Inverse transformation

The model transformation is not injective: it is not possible for example to get the classes A or C from the output model. This oracle can not be used here and according to our experience this is often the case.

4.1.3. Expected model

The tester should produce the expected models of the Figure 3 and Figure 5. They are as complicated as the test model and obviously as the output models since they check their entire validity. We notice that they contain many useless concepts (attributes, types, keys) when considering only the rule studied. It is important to notice that this requires a large effort compared to the simplicity of the evolution of the specification. This oracle function implies doubling of the complexity and the effort to write the new expected model while the evolution from one version of the transformation to another is simple.

4.1.4. Generic contract

It is possible to write a generic contract in OCL (an equivalent can be written in Kermeta for instance):

```
post table_correctly_created :
result.table.size=inputModel.classifier
 .select(cr|cr.oclIsTypeOf(Class))
 .select(c|c.oclAsType(Class).is_persistent)
 .select(cp|not cp.oclAsType(Class).parents
       .exists(p | p.is_persistent)).size
and //note: the classes have different names
inputModel.classifier
 .select(cr|cr.oclIsTypeOf(Class))
 .select(c|c.oclAsType(Class).is_persistent)
 .select(cp|not cp.oclAsType(Class).parents
       .exists(p | p.is_persistent))
 .forAll(csp|result.table
       .exists(t |t.name = csp.name))
```

This contract is not very complex, but the rule considered is one of the simplest. It is possible to write it differently especially with another contract language. But we can notice than the navigations and selections are quite repetitive. To test all the transformation, a set of generic contracts has to be written to consider all the requirements. We needed 14 contracts in total to completely specify the transformation. This contract can be reused for the new version of the transformation by removing the two bold select(…). The other contracts also have to be adapted in such a way, even if they do not consider this specific rule.

In [12], we pointed out a limitation of OCL as the language for expressing contracts. When contracts become too complex, they are difficult to express and maintain, and this may lead to the introduction of faults in these contracts.

4.1.5. Model snippets

The Figure 6 represents five model snippets based on the RDBMS metamodel. MF1 to MF4 define only one named table, and MF5 with an unnamed table. So, these snippets can be used in oracle function where one wants to assert the presence of a table named B (with MF1), or A (with MF2), or C (with MF3), or D (with MF4), or to assert the presence of a table that has no name (with MF5). We use these model snippets to write several oracles:

```
o1: {(MF1 , 1 , =)}
o2: {(MF2 , 0 , =) , (MF3 , 0 , =)}
o3: {(MF4 , 0 , =)}
o4: {(MF5 , 1 , =)}
```

We use these oracles to check the validity of the output model (Figure 3) returned by the transformation of the test model Mt1 (Figure 2) considering the rule `Ru`.

The oracle o1 validates "The persistent classes…are transformed into tables with same names": since there is a persistent class B in the test model, o1 checks that there is a table named B in the output model. o2 validates "and only these ones": since there are two non persistent classes in Mt1, o2 checks that there is no table with the same names. o3 validates "except if they inherit directly or not from another persistent class": since there is a persistent class D which inherits from the persistent class B, o3 checks there is no table named D in the output model. Finally o4 validates also "and only these ones" but without specifically considering the attributes `persistent`; o4 checks that only one table is created in the output model.

With these four oracles, four test cases can be written using the same test model.

These model snippets and their oracles are quite simple to write and to modularize depending on the rule considered. It is easy to reuse them with the new version: o1 and o2 are the same, and o3 is adapted by changing the cardinality to 1 because all the persistent classes even with a persistent parent are transformed and o4 with the cardinality 2 because the output model should only contain 2 tables.

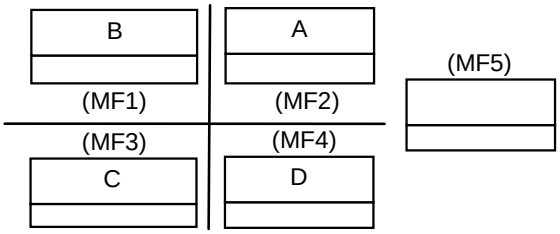

Figure 6. Five RDBMS model snippets

4.1.6. OCL Assertions

The patterns of the four previous oracles (4.1.5) can be implemented with OCL assertions. For instance the second pattern could be:

```
result.table.select(t|t.name=A).size()=0 and
result.table.select(t|t.name=C).size()=0
```

These constraints can also be easily reused but we think they are less easy to write. Even if learning a constraint language as OCL is not a complicated task for the tester, write patterns in the same way than test model is an advantage.

5. First analysis

In this section, we make a first analysis of these six oracle functions depending on the context the testers will use them in their test cases for model transformations. The context can differ depending on the complexity of the model transformation under test, of its output models, on the reuses and evolutions of this model transformation.

A high ***complexity*** of the model transformation under test is an obstacle to oracles using reference or inverse model transformations. As an inverse transformation is as complicated as the model transformation, the tester will have difficulties to get or develop trustable versions of them (sometimes it is even impossible). With a complex model transformation, it is necessary to use oracles which do not consider all the requirements at the same time. This is the case of the patterns (model snippets, OCL assertions), or of the generic contracts and even of the expected models.

A high complexity of the output models returned by the model transformation makes it difficult to use oracles which check the validity of an entire model at once. In this case the expected models and their oracle function should not be used.

When many test models are necessary, at least so many test cases are created. To reduce the effort and the risk of making an error, it is necessary that each test case does not have its own oracle, but that an oracle is reused in different test cases. Such oracle is generic and not dedicated to a test case, its test model, and its corresponding output model. Oracle functions using patterns (model snippets, assertions) or expected models are not adapted since they need the writing of at least one oracle data for each test case. Generic oracle data are preferable since they are written only once, and could be used with their corresponding oracle function in any test case. Reference, inverse transformations and generic contracts are generic oracle data that could be used in the case of numerous test cases.

Finally we consider the reuse and evolution of the model transformations. An oracle data and its oracle function will be reused in the test cases of the new version if the changes in the specification do not affect them. Then oracles which check the validity of the output models with respect to many requirements will have more chance to be affected by the slightest change in the specification. Therefore, we would not advise the tester to use the reference and inverse transformation because they will be affected by any change of the specification. Many expected models

could also need adaptation if the specification impacts a part of the implementation that is often used during the test models transformations (as in our example). The generic contracts have a comparable problem. A contract could be made with several repetitive navigations and filterings. Even if a generic contract is dedicated to check few requirements, it could contain such navigation and filtering that could be affected by the specification changes. The patterns do not have these problems since they check only a few requirements and are dedicated to test cases and their input model. They will be a little affected by specification changes.

6. Conclusion

We have presented the issues that a tester will face during the specification of model transformation testing oracles. The complexity of the model transformation and the data they are manipulated, the models, raise the complexity to write oracles. We presented several oracle functions that can be used in test cases and we provide several advices to choose a function despite another one depending on the complexity of these output models and model transformations, and how these lasts will be reused.

In further work, it will be necessary for us to provide strict criteria to completely measure the advantages and drawbacks of each oracle function depending on their oracle data.

7. References

1. OMG. *MOF 2.0 Core Final Adopted Specification.* 2004 2005; Available from: http://www.omg.org/cgi-bin/doc?ptc/03-10-04.
2. Bezivin, J., B. Rumpe, A. Schurr, and L. Tratt, *Model Transformations in Practice Workshop*. 2005: Supplemental Proceedings of the MoDELS'05 conference. p. 120 - 127.
3. Fleurey, F., B. Baudry, P.-A. Muller, and Y. Le Traon, *Towards Dependable Model Transformations: Qualifying Input Test Data.* Software and Systems Modeling, 2007.
4. Ehrig, K., J.M. Küster, G. Taentzer, and J. Winkelmann. *Generating Instance Models from Meta Models*. in *FMOODS'06 (Formal Methods for Open Object-Based Distributed Systems)*. 2006. Bologna, Italy.
5. Sen, S., B. Baudry, and J.-M. Mottu, *On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing*, in *Proceedings of ICST'08 (International Conference on Software Testing Verification and Validation)*. 2008: Lillehammer, Norway.
6. Lin, Y., J. Gray, and F. Jouault, *DSMDiff: A Differentiation Tool for Domain-Specific Models.* European Journal of Information Systems, Special Issue on Model-Driven Systems Development, 2007.
7. Heckel, R. and M. Lohmann, *Towards Model-Driven Testing.* Electronic Notes in Theoretical Computer Science, 2003. **82**(6).
8. Alanen, M. and I. Porres. *Difference and Union of Models*. in *UML'03 (Unified Modeling Language)*. 2003. San Francisco, CA, USA.
9. *EMF Compare*. Eclipse Foundation; Available from: www.eclipse.org/emft/projects/compare.
10. Le Traon, Y., B. Baudry, and J.-M. Jézéquel, *Design by Contract to Improve Software Vigilance.* IEEE Transactions on Software Engineering, 2006. **32**(8).
11. Briand, L.C., Y. Labiche, and H. Sun, *Investigating the Use of Analysis Contracts to Improve the Testability of Object Oriented Code.* Software Practice and Experience, 2003. **33**(7).
12. Mottu, J.-M., B. Baudry, and Y. Le Traon. *Reusable MDA Components: A Testing-for-Trust Approach*. in *MoDELS'06*. 2006. Genova, Italy.
13. de Lara, J. and H. Vangheluwe, *AToM3: A Tool for Multi-formalism and Meta-modelling*, in *FASE '02 (International Conference on Fundamental Approaches to Software Engineering)*. 2002.
14. Kolovos, D.S., R.F. Paige, and F.a.C. Polack. *Model Comparison: A Foundation for Model Composition and Model Transformation Testing*. in *workshop GaMMa'06*. 2006. Shangaï, China.
15. Küster, J.M. and M. Abd-El-Razik. *Validation of Model Transformations - First Experiences using a White Box Approach*. in *workshop MoDeV²a 2006, colocated with MoDELS06*. 2006. Genova, Italy.
16. Ramos, R., O. Barais, and J.-M. Jézéquel. *Matching Model-Snippets*. in *MoDELS'07*. 2007. Nashville, USA.
17. Muller, P.-A., F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, and P. Studer. *On Executable Meta-Languages applied to Model Transformations*. in *Model Transformation in Practice Workshop, part of MoDELS'05*. 2005. Montego Bay, Jamaica.
18. Lawley, M. and J. Steel, *Practical Declarative Model Transformation With Tefkat*, in *Model Transformation in Practice Workshop, part of MoDELS'05*. 2005: Montego Bay, Jamaica.