Testing M2M/M2T/T2M Transformations

Loli Burgueño
Dept. Lenguajes y Ciencias de la Computación
University of Malaga
Malaga, Spain
Email: loli@lcc.uma.es

***Abstract*—As Model-Driven Engineering is becoming adopted by industry, models and model transformations (MTs) are extensively used. Hence, there is the urgent need for systematic testing mechanisms and tools to check their correctness. In this work, we make use of a particular case of contracts for model transformations called Tracts. First, Tracts allow the transformation developer to specify and test a model-to-model transformation in a modular way, and to identify bugs. However, they do not allow to track where the faults in the implementation are. For doing that, we present an approach based on matching functions that automatically establish the alignments between the specification and the implementation of a transformation using the metamodel footprints. Second, we extend Tracts to deal with text-to-model and model-to-text transformations in order to broaden and complete the scope of our testing proposal. Finally, we provide the corresponding tools that realize our proposal.**

I. Problem and Motivation

Model transformations (MT) are gaining more and more interest as industry is progressively adopting model-driven techniques [1]. The main advantage of using of model transformations is to save effort and reduce errors by automating the creation and alteration of models as long as it is possible. Thus, they are becoming a promising approach in many different scenarios to solve a wide variety of problems, e.g. to deal with the migration of systems, their modernization, for code generation, etc., especially when complex data structures are involved. This complexity may lead to the existence of bugs in the model transformation implementation that make it faulty. Then, the need of testing, validation and verification procedures for model transformations is emerging in recent years [2], [3].

So far, most of the efforts by the research community have focused on testing model-to-model (M2M) transformations for which having explicit model representations for the input and output domain is assumed. There are different approaches that can be classified attending to their characteristics as black-box vs. white-box and static vs. dynamic. Depending on the concrete situation, the transformation developer needs to make the decision of what mechanism to use. When black-box dynamic approaches such as Tracts [4] are the best option, the developer finds that they do allow the testing of model transformations but they do not track where the problem is in the implementation, i.e., they reveal that there is a problem but they do not point to where it is or what is producing it.

Furthermore, text-to-model (T2M) and model-to-text (M2T) transformations are extensively used [5] for code generation and reverse engineering for the modernization of legacy applications [6]. However, they have received little attention so far by the research community.

The contribution presented in this paper is twofold. First, we have extended the Tract approach for M2T and T2M transformations. We have created a generic metamodel that represents text repositories and inject the text into a model that conforms to that metamodel. Once both source and target domains count on a concrete and well-defined representation, M2T and T2M transformations are reduced to M2M transformations. Therefore, Tracts can be used for checking their correctness.

Second, we define a mechanism based on *matching tables* that permit relating the rules of a model transformation with its Tracts, i.e., aligning the model transformation implementation with its specification. By analysing the matching tables, the rules that cause a fault can be identified, hence realizing a useful tracking mechanism for locating faults in model transformations.

The structure of the paper is as follows. Section II introduces the concepts in which this work stands and the related work. Section III presents the core of our contribution: the extension of Tracts for M2T and T2M transformations and how the matching tables are computed and interpreted. Finally, Section IV shows the results we have obtained and the contributions we have made.

II. Background and Related Work

The need for systematic verification of model transformations has been studied in previous works and the challenges it has to deal with have been outlined [7], [8]. Many approaches ranging from lightweight certification to full verification have been proposed to reason about different kinds of properties of M2M transformations [2], [3]. One of them is the use of contracts [4], [9], [10].

Tracts, which are a particular case of contracts, are a black-box testing mechanism for M2M transformations. They consist of a set of constraints on the source and target metamodels, a set of source-target constraints, and a test suite, i.e., a collection of source models. They provide modular pieces of specification, each one focusing on a particular transformation scenario. This permits each model transformation to be specified by means of a set of Tracts, each one covering a specific use case. Usually, they are seen as unit tests, which means that developers identify the scenarios of interest and define a Tract

for each one. Then, they check whether the transformation behaves as expected in these scenarios. Other works proposed alternative ways for defining oracles [11]–[15], however, they do not discuss how to apply their approaches for text artifacts. The most closely related work for testing M2T transformations is presented in [16]. Nevertheless, this approach requires the definition of a functional decomposition diagram for the M2T transformation as well as the design specifications for the text produced by the transformation.

Tracking guilty transformation rules using a dynamic approach where constraints are involved has also been subject to investigations [17], [18]. In [19]–[21], the authors locate errors using the trace information of the MT executions, i.e., determining the relationship between the source and target elements and the excerpt of the transformation involved. The dynamic approach is also used in [22] to build slices of model transformations and in [23] following a white-box testing approach. All these approaches need to count on input models while our aim is to statically build more general traceability models between the specification of the MTs and their implementations.

III. Approach and Uniqueness

A. Extending Tracts for M2T and T2M transformations

In order to test model transformations when text is involved in one of the domains, either in the source or the target, we propose an approach that converts the problem to a M2M transformation testing problem [24]. In order to achieve that, instead of defining a specific grammar or metamodel for each text artifact, we have opted for creating a generic metamodel that represents text repositories. Then, the folder structure and text files are injected to a model conforming to the text metamodel.

The metamodel is shown in Figure 1. It counts on a metaclass `Repository` that represents the entry point to the root folder containing folders and files or to a file if only one single artifact is used. Folders just contain a name while files have in addition an extension as well as a content. The content of files is represented by lines that are sequentially ordered. A derived attribute content is used to allow easy access to the complete content of a file.

Figure 2 displays on its left-hand side the folder structure of a Java project while on its right-hand side the content of one of its Java files. Figure 3 presents an excerpt of the text model corresponding to the elements that Figure 2 shows and several lines of the Java file.

The specification of the transformation to be tested is composed of a set of Tracts, each one focusing on a particular property that the developer wants to ensure. The constraints defined by those Tracts are OCL expressions. Thus, a problem arises when the developer needs to deal with the text represented by the lines but realises that the variety of libraries and operations that OCL provides to manage Strings is reduced and very restrictive. As the text in the lines may need to be

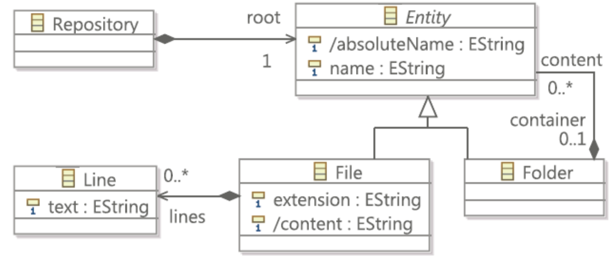

Fig. 1. Metamodel for representing text artifacts and repositories.

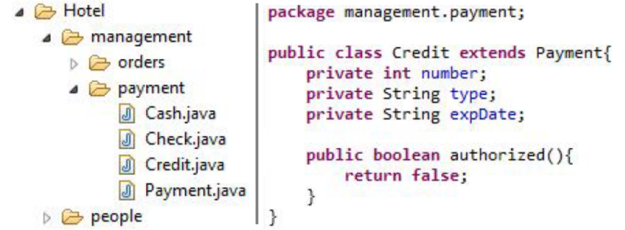

Fig. 2. Exemplary folder structure and file content.

analysed thoroughly, we have enriched OCL with an operation called `matchesRE()` that checks whether a given string matches a regular expression. Furthermore, we have introduced some auxiliary functions that are currently provided by M2T transformation languages such as `toFirstUpper()` to end up with more concise OCL constraints than just using the standard OCL String operation library.

In order to illustrate what a Tract looks like, let us assume that we are testing a M2T transformation that generates Java code from UML models. Let us also assume a very simplified UML metamodel that only has `Packages`, `Classes` and `Properties`. All of them have a name and the properties have a type as well. Furthermore, each package may contain a set of classes and each class may contain set of properties. The metamodel is shown in Figure 4.

The Tract constraint in Listing 1 specifies the correct behavior that a M2T transformation that transforms each UML package to a Java package, each UML class to a Java class and each UML property to a Java attribute must fulfil.

Listing 1. Tract constraint for the *UML2Java* example.

```
UMLPackage.allInstances->forAll(upack |
  Folder.allInstances->exists(folder |
    upack.name = folder.name and
    upack.classes->forAll( uclass |
      folder.content->selectByType(File)->exists( file |
        uclass.name = file.name and
        uclass.properties.allInstances->forAll(uprop |
          file.lines->exists( line | line.machtesRE(
            ".*"+uprop.type+".*"+uprop.name+".*;")))))))
```

In order to provide tool support for our proposal, we have developed a injector (parser) that converts the content of a text repository into a model that conforms to the text metamodel shown in Figure 1, and an extractor that takes models conforming to the text metamodel and produces text organized in folders. In order to check that a given M2T transformation fulfils the constraints (such as the one shown

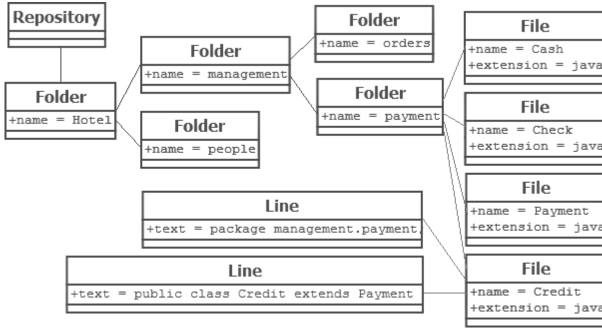

Fig. 3. Text Model

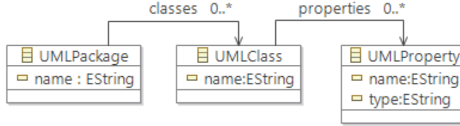

Fig. 4. Simplified UML Metamodel

before), we execute the transformation using the Tract test suite and then, we use the injector for obtaining the output models conforming to the text metamodel from the output text generated by the transformation. Then we check the validity of the constraints as in the case of Tracts defined for M2M transformations, with our TractsTool [3]. TractsTool evaluates the defined constraints on the source and target models by a transparent translation to the USE tool [25].

In the case that the MT to test is a T2M transformation, the procedure is similar. The test suite is defined by the Tract as a set of repositories, which need to be transformed first into a model-based representation using our injector. When the source constraints are fulfilled, the content of the repository is transformed by the T2M transformation under test to produce the output models. The models produced from the repository and their corresponding output models can then be validated by TractsTool against the Tracts.

In this way, we are able to test M2T and T2M transformations in a similar manner to M2M transformations.

B. *Tracking faults in MT implementations*

Using Tracts in conjunction with the previously presented approach, M2M, M2T and T2M transformations can be tested but once a failure is detected, it is not possible to track why the transformation is not working or where the problem is located. The existence of a problem lies in the misalignment between the model transformation specification and its implementation. We present a white-box and static analysis to ease the task of finding the location of the model transformation rules that may have caused the faulty behavior [26].

We are using Tracts for defining the specification of the MTs. ATL [27] is the MT language we have chosen for building its implementatation. ATL is a rule-based language containing a mixture of declarative and imperative constructs for defining uni-directional transformations. A rule consists of an *input* pattern—that might have a *filter* condition or not—which is matched on the source model, and an *output* pattern that produces certain elements in the target model for each match of the input pattern. OCL expressions are used to calculate the values of target features of the target elements.

Given the set of OCL constraints from the Tracts and the set of ATL rules from the transformation implementation, the common part they share is the source and target metamodels, which means that the same types and features are used. Thus, we make use of these commonalities to indirectly match the constraints and the rules by matching their footprints concerning the source and target metamodels. Our aim is to construct three tables called *matching tables* with the alignments between specification and implementation.

First of all, we need to extract the footprints (i.e., the structural elements) from the constraints and rules. Since metamodels are graphs, OCL expressions are heavily dependent on their *contexts* and also on the path used to navigate to the types that the constraint is checking. That navigation path has nothing to do with the aim of the constraint. Thus, taking all the footprints on it into account only introduces noise. This is why we only consider as relevant the last elements of the OCL expressions. To consider operations on collections, we take into account only the footprints inside the body of the deepest (in the sense of nesting) iterators (*forAll*, *exists*, etc.).

Once the footprints have been extracted, for each pair constraint/rule, the percentage of overlapping footprints is calculated. To do so, we also take subtyping into account, i.e., we consider that two footprints matches if they share the same type or if one is a subtype of the other. This is important because some OCL operators used in the Tract constraints and in the ATL rules (such as *allInstances*) retrieve all instances of a certain class, as well as the instances of all its subclasses. When the information about the footprints is available, the matching tables are calculated according to three metrics: *constraint coverage* (CC), *rule coverage* (RC) and *relatedness of constraints and rules* (RCR). The value for the cell $[i, j]$ is given by the following formulas:

$$CC_{i,j} = \frac{|C_i \cap R_j|}{|C_i|}; RC_{i,j} = \frac{|C_i \cap R_j|}{|R_j|}; RCR_{i,j} = \frac{|C_i \cap R_j|}{|C_i \cup R_j|}$$

CC measures the coverage for constraint i by a given rule j. We interpret this value for rule traceability, i.e., to find the rules related to the given constraint. This is, if a constraint fails, the CC table tells us which rule or rules are more likely to have caused the faulty behavior. For this reason, the CC table is to be consulted by rows.

On the other hand, RC focuses on rules. This metric calculates the coverage for rule j by a given constraint i. We use the RC table to express constraint traceability as it shows what constraints are more closely related to a given rule. Therefore, it is to be read by columns.

RCR is related to both constraints and rules so it can be consulted by rows and by columns. It provides information

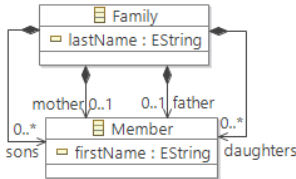

Fig. 5. The Family and Person metamodels.

about the relatedness of both rules and constraints, without defining a direction for interpreting the values.

Let us use the well-known M2M transformation example *Families2Persons*[1] to show how our approach is applied. The two metamodels are presented in Figure 5. We have developed one Tract (Listing 2) that considers only families with exactly four members: one mother, one father, one daughter and one son. The first constraint states that all families in the source model have exactly one daughter and one son. The second constraint states that all mothers and daughters are transformed into female persons and the third mandates that all fathers and sons are transformed into male persons. Finally, the last constraint checks that the sizes of the source and target models are equal.

Listing 2. Tracts for the *Families2Persons* case study.

```
-- C1: SRC_oneDaughterOneSon
Family.allInstances->forAll(f|f.daughters->size=1 and
  f.sons->size=1)
-- C2: SRC_TRG_MotherDaughter2Female
Family.allInstances->forAll(fam|Female.allInstances->
 exists(f|fam.mother.firstName.concat('␣').concat(
 fam.lastName)=f.fullName) xor fam.daughters->exists(d|
  d.firstName.concat('␣').concat(fam.lastName)=f.fullName))
-- C3: SRC_TRG_FatherSon2Male
Family.allInstances->forAll(fam|Male.allInstances->
 exists(m| fam.father.firstName.concat('␣').concat(
 fam.lastName)=m.fullName xor fam.sons->exists(s|
  m.firstName.concat('␣').concat(fam.lastName)=s.fullName))
-- C4: SRC_TRG_MemberSize_EQ_PersonSize
Member.allInstances->size=Person.allInstances->size
```

The footprints extracted for C1 are *Family*, *Family.daughters*, *Family.sons* and *Member* (*Member* appears because it is the type of *f.daughters* and *f.sons*). For C2, they are *Family*, *Member*, *Female*, *Member.firstName*, *Family.lastName*, *Female.fullName*. Note that, in the case of the navigation path *Family.mother.firstName*, we will only consider *mother.firstName*. The footprints for C3 are the same as for C2 but replacing *Female* with *Male*. Finally, the footprints for C4 are *Member* and *Person*.

A possible implementation of the transformation in ATL is given in Listing 3. It comprises two helper functions and two rules. One of the helpers is used to decide whether a member is female or not, and the second one is used to compute the family name of a family member. The first rule, R1, transforms male members (note the use of the helper isFemale() to filter the corresponding source objects) into male persons and R2 is analogous, but for female family members.

Listing 3. *Families2Persons* ATL Transformation.

[1]http://www.eclipse.org/atl/atlTransformations/#Families2Persons

TABLE I
Families2Persons MATCHING TABLES.

	CC		RC		RCR	
	R1	**R2**	**R1**	**R2**	**R1**	**R2**
C1	0.33	0.33	0.25	0.25	0.17	0.17
C2	0.33	0.67	0.50	1.00	0.25	0.67
C3	0.67	0.33	1.00	0.50	0.67	0.25
C4	1.00	1.00	0.50	0.50	0.4	0.4

```
module Families2Persons;
create OUT: Persons from IN: Families;
helper context Families!Member def:isFemale:Boolean=...
helper context Families!Member def:familyName:String=...
rule Member2Male { -- R1
 from s:Families!Member (not s.isFemale)
 to t:Persons!Male(fullName<-s.firstName+'␣'+s.familyName)}
rule Member2Female { -- R2
 from s:Families!Member (s.isFemale)
 to t:Persons!Female(fullName<-s.firstName+s.familyName)}
```

The footprints corresponding to R1 are *Member*, *Male*, *Member.firstName* and *Male.fullName*, while the footprints for R2 are *Member*, *Female*, *Member.firstName* and *Female.fullName*. Note that when a rule calls a helper, the helper's footprints are included into the set of rule's footprints.

Table I shows the metrics computed for the *Families2Persons* example. Note that, for a small example like this, the metrics provide information that can be easily interpreted by just looking at the constraints and the rules. Let us suppose that we have executed the transformation for a certain input model and checked the satisfaction of the constraints using TractsTool. Let us assume the outcome given by the tool is that constraint C2 is not satisfied. Looking at the CC metric, we can see that it is more likely that the problem is in rule R2. In case there were several rules with the same probability in CC, we should look at RCR to decide which one should be checked first. By looking at RC, we can see that for each one of the rules, there is always a constraint covering it, what means that it correctness is being checked by the Tract.

The testing method we propose is far from fully prove correctness of the transformation. Nevertheless, it provides the first step to model transformation testing, which aims at locating faults as early as possible in a quick and cost-effective manner. Being aware that our proposal may not work in some cases, we also provide a method and a tool, called Similarity Matrix Calculator, for checking whether a transformation is amenable to be used with it. A similarity matrix gives us an indication of how rules are related with each other looking at the common footprints they share. We obtain the mean and the standard deviation of the rule similarities. The lower both values are (especially the mean), the fewer types and features the rules have in common, and thus, the higher the chance for a successful application of our approach is. However, if the mean and the standard deviation are far from 0, it is difficult to distinguish among the rules which is the "guilty" one.

All the previously mentioned tool support we have developed, i.e., TractTool, Matching Tables Builder and Similarity Matrix Calculator, can be downloaded from our website[2].

[2]http://atenea.lcc.uma.es/index.php/Main_Page/Resources/FaultLocMT

TABLE II
EVALUATION RESULTS

TOOL	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
ArgoUML	✓	✓	×	×	-	✓	✓	×	×
Poseidon	✓	×	×	✓	×	✓	✓	×	✓
MagicD.	✓	✓	✓	✓	×	✓	✓	×	✓
EArchitect	✓	✓	✓	×	×	✓	✓	×	×
BOUML	×	✓	-	✓	×	✓	✓	×	✓
A.UModel	×	✓	✓	✓	×	✓	✓	✓	✓

IV. RESULTS AND CONTRIBUTIONS

In order to evaluate the usefulness of our contract-based mechanism for M2T or T2M transformations, we have selected a set of currently available UML tools that provide code generation facilities to produce source code from UML models. We have generated the Java code corresponding to a set of input models and we have checked the result using a set of Tracts.

For the evaluation, we defined a set of 9 constraints which represent some of the most essential requirements that any UML to Java code generator has to fulfil. C1 establishes that nested packages are transformed into nested folders. C2 checks that the import of packages is supported and C3 that the inheritance of a leaf class is not allowed. C4 makes sure that only single inheritance is used in UML and C5 that derived attributes only result in getter methods. C6 and C7 checks that the visibility of attributes and roles is mapped to Java. C8 watches that no Java keywords are allowed as names in UML models and C9 that the names in an UML model have to be valid Java identifiers. The constraints as well as all the required files to execute the experiment can be found in our website [3].

The six UML tools that we selected from industry claimed to support code generation from UML class diagrams into Java code. The selected sample covers both commercial tools and open-source projects. They are ArgoUML v.0.34, Poseidon v.6.0.2., MagicDraw v.16.8., EnterpriseArchitect v.10, BOUML v.4.22.2. and Altova UModel.

We defined reference test models based on UML and we re-modelled them in all of the selected tools. We run the code generator for each one of the tools and obtainted the Java text corresponding to the UML model. Then we checked the output against the Tracts.

Table II shows the results of the evaluation. A tick symbol (✓) means that the test passed for that Tract and a cross symbol (×) means that the Tract test failed. Some of the tests were not available for a given tool, e.g., a particular modeling feature is missing, and were not performed. This is indicated by a dash (-). We found that no tool performs well even with respect to the basic UML to Java code generators. Furthermore, we discovered that several tools produced incorrect Java code, even not compilable in some situations. In this sense, the Tracts presenting the basic requirements could be used as the initial components of a benchmark for future improvements and developments of UML-to-Java code generators.

In order to evaluate the second part of our proposal—where, given the failures in the Tracts, the rules that cause the problems are disclosed—we have analyzed the alignments between specifications and implementations in four different real-world transformation projects. For each one of them we computed manually the alignments between rules and constraints. Having the matching tables obtained with our approach and the real alignments, we are able to compute two measures to assess its quality: *precision* and *recall*. In the context of our study, precision denotes the fraction of the detected alignments that are in fact correct. Recall indicates the fraction of alignments that have not been missed.

The first case study we selected is a transformation dealing with the generation of Entity Relationship (ER) Diagrams from UML Class Diagram Models. Second, a project that deals with behavioral models conforming to CPL [28] that are transformed into models conforming to SPL [29]. This transformation [30] is a relatively complex example available from the ATL zoo[4]. Also from the ATL zoo, we considered a project that does not operate on modeling languages but rather on markup languages. It is the *BT2DB* transformation from BibTeX documents into DocBook documents. Finally, we experimented with a very large transformation called *Ecore2Maude* which is used by the tool e-Motions [31] in order to apply some formal reasoning.

For space reasons, we cannot present the Tract constraints, the ATL rules and the matching tables but they are available at our website [5]. We have observed that the values obtained for the precision and recall metrics are acceptable in three of the projects: *UML2ER*, *CPL2SPL* and *Ecore2Maude*. With these accuracy results, we can conclude that our approach works well, since the alignments found statically are quite reliable. Nevertheless, as pointed by the similarity matrix for the *BT2DB* example (with a mean of 0.41 and a standard deviation of 0.24) our approach is unable to help detect problems in the implementation. We have also computed the similarity matrixes for all the transformations in the ATL zoo in order to investigate the applicability of our approach. Out of the 41 model transformations studied, the mean and standard deviation turned out to be below 0.15 in 21 of them, which means that our approach can be used with around half of the transformations.

We conclude this paper by listing the contributions we have made. First, we have extended Tracts to be used for M2T/T2M transformations and have proved its usefulness detecting errors in current UML-to-Java code generators offered by well-known UML tools. Second, we have presented a static approach to trace errors in model transformations and have proved that our approach is applicable to a large number of transformations.

[3]http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Tracts/UML2Java

[4]http://www.eclipse.org/atl/atlTransformations

[5]http://atenea.lcc.uma.es/index.php/Main_Page/Resources/MTB

REFERENCES

[1] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, ser. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[2] M. Amrani, L. Lúcio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. L. Traon, and J. R. Cordy, "A tridimensional approach for studying the formal verification of model transformations," in *Proc. of the 1st International Workshop on Verification and Validation of Model Transformations (VOLT 2012)*, 2012.

[3] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann, "Formal specification and testing of model transformations," in *Formal Methods for Model-Driven Engineering (SFM)*, ser. LNCS, vol. 7320. Springer, 2012, pp. 399–437.

[4] M. Gogolla and A. Vallecillo, "*Tract*able model transformation testing," in *Proc. of ECMFA'11*, ser. LNCS, vol. 6698. Springer, 2011, pp. 221–236.

[5] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–646, 2006.

[6] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: a generic and extensible framework for model driven reverse engineering," in *Proceedings of the 25th International Conference on Automated Software Engineering (ASE 2010)*. ACM, 2010, pp. 173–174.

[7] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. L. Traon, and J.-M. Mottu, "Barriers to systematic model transformation testing," *Communications of the ACM*, vol. 53, no. 6, pp. 139–143, 2010.

[8] R. V. D. Straeten, T. Mens, and S. V. Baelen, "Challenges in model-driven software engineering," in *Models in Software Engineering*, ser. LNCS, vol. 5421. Springer, 2008, pp. 35–47.

[9] B. Meyer, "Applying design by contract," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[10] E. Cariou, N. Belloir, F. Barbier, and N. Djemam, "OCL contracts for the verification of model transformations," *ECEASST*, vol. 24, 2009.

[11] E. Cariou, R. Marvie, L. Seinturier, and L. Duchien, "OCL for the specification of model transformation contracts," in *Proc. of the OCL and Model Driven Engineering Workshop*, 2004.

[12] 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 *Proc. of MODELS'11*, ser. LNCS, vol. 6981. Springer, 2011, pp. 395–409.

[13] P. Giner and V. Pelechano, "Test-Driven Development of Model Transformations," in *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, ser. LNCS, vol. 5795. Springer, 2009, pp. 748–752.

[14] D. Kolovos, R. Paige, L. Rose, and F. Polack, "Unit testing model management operations," in *ICSTW'08*. IEEE, 2008, pp. 97–104.

[15] E. Guerra, "Specification-driven test generation for model transformations," in *Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT 2012*, ser. LNCS, vol. 7307. Springer, 2012, pp. 40–55.

[16] A. Tiso, G. Reggio, and M. Leotta, "Unit testing of model to text transformations," in *Proc. of AMT'14*, 2014, pp. 14–23. [Online]. Available: http://ceur-ws.org/Vol-1277/2.pdf

[17] B. Ramesh and V. Dhar, "Supporting systems development by capturing deliberations during requirements engineering," *IEEE Transactions on Software Engineering*, vol. 18, no. 6, pp. 498–510, 1992.

[18] F. A. C. Pinheiro and J. A. Goguen, "An object-oriented tool for tracing requirements," *IEEE Software*, vol. 13, no. 2, pp. 52–64, 1996.

[19] M. Hibberd, M. Lawley, and K. Raymond, "Forensic debugging of model transformations," in *Proc. of MODELS'07*, ser. LNCS, vol. 4735. Springer, 2007, pp. 589–604.

[20] V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser, "Traceability mechanism for error localization in model transformation," in *Proc. of ICSOFT'09*. INSTICC Press, 2009, pp. 66–73.

[21] M. Wimmer, G. Kappel, J. Schönböck, A. Kusel, W. Retschitzegger, and W. Schwinger, "A Petri Net based debugging environment for QVT Relations," in *Proc. of ASE'09*. IEEE, 2009, pp. 3–14.

[22] Z. Ujhelyi, Á. Horváth, and D. Varró, "Dynamic backward slicing of model transformations," in *Proc. of ICST'12*. IEEE, 2012, pp. 1–10.

[23] C. A. González and J. Cabot, "ATLTest: A White-Box Test Generation Approach for ATL Transformations," in *Proc. of MoDELS'12*, ser. LNCS, vol. 7590. Springer, 2012, pp. 449–464.

[24] M. Wimmer and L. Burgueño, "Testing M2T/T2M transformations," in *Proc. of MODELS'13*, ser. LNCS, vol. 8107. Springer, 2013, pp. 203–219.

[25] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Science of Computer Programming*, vol. 69, pp. 27–34, 2007.

[26] L. Burgueño, M. Wimmer, J. Troya, and A. Vallecillo, "Static Fault Localization in Model Transformations," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 490–506, 2015.

[27] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[28] J. Lennox, X. Wu, and H. Schulzrinne, "Call Processing Language (CPL): A language for user control of internet telephony services," 2004, http://www.ietf.org/rfc/rfc3880.txt.

[29] L. Burgy, C. Consel, F. Latry, J. Lawall, N. Palix, and L. Reveillere, "Language technology for internet-telephony service creation," in *Proc. of ICC'06*. IEEE, 2006, pp. 1795–1800.

[30] F. Jouault, J. Bézivin, C. Consel, I. Kurtev, and F. Latry, "Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages," in *Proc. of ECOOP Workshop on Domain-Specific Program Development*, 2006.

[31] J. E. Rivera, F. Durán, and A. Vallecillo, "A Graphical Approach for Modeling Time-Dependent Behavior of DSLs," in *Proc. of VL/HCC'09*. IEEE, 2009, pp. 51–55.