Employing Classifying Terms for Testing Model Transformations

Martin Gogolla
University of Bremen
Bremen, Germany
gogolla@informatik.uni-bremen.de

Antonio Vallecillo
University of Málaga
Málaga, Spain
av@lcc.uma.es

Loli Burgueño
University of Málaga
Málaga, Spain
loli@lcc.uma.es

Frank Hilken
University of Bremen
Bremen, Germany
fhilken@informatik.uni-bremen.de

***Abstract*—This contribution proposes a new technique for developing test cases for UML and OCL models. The technique is based on an approach that automatically constructs object models for class models enriched by OCL constraints. By guiding the construction process through so-called classifying terms, the built test cases in form of object models are classified into equivalence classes. A classifying term can be an arbitrary OCL term on the class model that calculates for an object model a characteristic value. From each equivalence class of object models with identical characteristic values one representative is chosen. The constructed test cases behave significantly different with regard to the selected classifying term. By building few diverse object models, properties of the UML and OCL model can be explored effectively. The technique is applied for automatically constructing relevant source model test cases for model transformations between a source and target metamodel.**

I. Introduction

As the complexity of model transformations grows, there is an increasing need to count on more powerful and precise testing techniques. One essential aspect of model transformation testing (and, in general, of software testing) is the selection of effective test cases [1].

One way to achieve this is by using *Equivalence Partitioning*, a software testing technique that divides the input data of a software unit into partitions of equivalent data from which test cases can be derived [2]. The fundamental concept of this technique is based on the use of equivalence classes, and the selection of one representative element from each class. An advantage of this approach is the reduction of the total number of test cases to a finite set of testable test cases, still covering a maximum of requirements. Testing time is also significantly reduced, due to lesser number of test cases.

The key idea of this approach is that we need to test only one input model from each partition as we assume that all the models in a certain partition will be treated in the same way by the transformation. If one model belonging to a partition has certain characteristics of interest, we assume all of the models in that partition will have them too and thus will behave the same. Therefore, there is no point in testing any of these others. Similarly, if one of the models in a partition does not work, then we assume that none of the models in that partition will work. Again, there is little point in testing any more in that partition. In sum, this is because all models in a partition are *equivalent*.

The main issues are how to define the equivalence classes that define the partitions in an expressive and flexible way, and how to automatically select one representative element of each class.

To achieve this, our contribution proposes a new technique for developing test cases for UML and OCL models, based on an approach that automatically constructs object models for class models enriched by OCL constraints. By guiding the construction process through so-called classifying terms, the built test cases in form of object models are classified into equivalence classes. Classifying terms are arbitrary OCL terms on a class model that calculate a characteristic value for each object model. Each equivalence class is then defined by the set of object models with identical characteristic values and with one canonical representative object model. By inspecting these object models, a developer can explore properties of the class model and its constraints.

In this contribution we also show how classifying terms can be effectively used in combination with Tracts [3], a specification and black-box testing approach for model transformations, providing a sound and practical mechanism for the automated generation of suitable test models for Tracts.

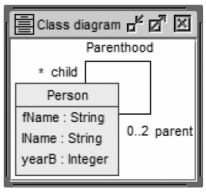
This paper is organized in 5 sections. After this Introduction, Section II introduces classifying terms, describes how they are specified, and presents the mechanism available for automatically constructing the representative object models. Then, Section III describes how classifying terms can be used in the context of Tracts to implement model transformation testing. Section IV relates our work to other similar approaches. Finally, Section V concludes and outlines some future lines of work.

II. Classifying Terms

Classifying terms are an instrument to explore model properties. We discuss their underlying concepts and their implementation in the context of a tool, the UML-based Specification Environment (USE). The underlying ideas can be employed however in similar modeling tools. USE allows the modeler to describe a system with a UML class model (class diagram) and OCL constraints, among other description means like, for example, UML protocol state machines. USE is intended for validation and verification of UML models. One central validation task is the automatic construction of object

models (object diagrams) for the class model including the OCL constraints. This task can be performed by a so-called model validator that (a) transforms UML and OCL models into the relational logic [4] of Kodkod [5], (b) analyzes the relational logic results, and (c) transforms the results back in terms of UML. The object model construction is guided by a configuration that specifies how classes, associations, attributes and data types are populated. Finite bounds must guarantee that all model elements (classes, associations, attributes and data types) are associated during the validation process with finite sets.

The running example in this section is a very simple `Parenthood` description as shown in Fig. 1 with a UML class model and accompanying OCL invariants. Given an appropriate configuration, the model validator can automatically construct object models like the ones in Fig. 2.




```
context p1,p2:Person inv nameUnique:
  p1<>p2 implies p1.fName<>p2.fName or
                 p1.lName<>p2.lName
context p:Person inv acyclicParenthood:
  p.parent->closure(parent)->excludes(p)
context p:Person inv parentOlderChild:
  p.child->forAll(c|p.yearB+15<=c.yearB)
```

Fig. 1. Example UML class model including OCL invariants.

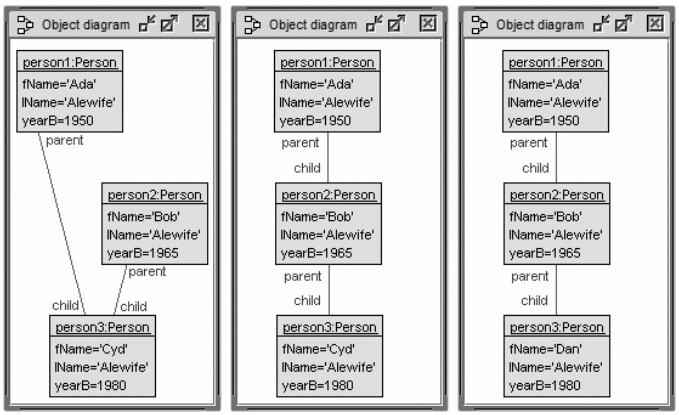


Fig. 2. Different example object models with partly isomorphic structure.

In order to explain the need for classifying terms, the central new notion in this contribution, let us consider the following model exploration task: for a given class model and under a particular configuration, the developer wants to *scroll* through *all* valid object models, i.e., she wants to consider not only a single object model but the collection of all valid object models. This is currently realized in the USE approach through the validation option *scrolling* that spans up all object models.

Problem: The general difficulty appearing now is that many very similar object models will be taken into account. The developer might expect to be shown *interesting*, structurally different object models. For example, in the above `Parenthood` model under a configuration requiring exactly three `Person` objects and two `Parenthood` links, the two rightmost object models in Fig. 2 will typically appear as distinct models, although being different only in the first name of the `Person` objects at the bottom. However, a development approach could offer the option to prevent that isomorphic object models with the same `Parenthood` patterns are presented as distinct object models, when scrolling through the collection of valid object models

Solution: As an answer, our approach gives the developer an explicit option to formulate her understanding of two object models being different. The technical realization is as follows: the developer specifies a closed OCL query term, i.e., a term without free variables, that can be evaluated in an object model and returns an (for the time being) integer as a characteristic value; in our approach, this term is called 'classifying term'; each newly constructed object model has to show a different characteristic value. As sketched in Fig. 3, the classifying term determines an equivalence relationship on all object models. Two object models with the same characteristic value belong into the same equivalence class. The approach decides to choose only one representative from each equivalence class. We will later lift the restriction that only one classifying term of type `Integer` is considered.

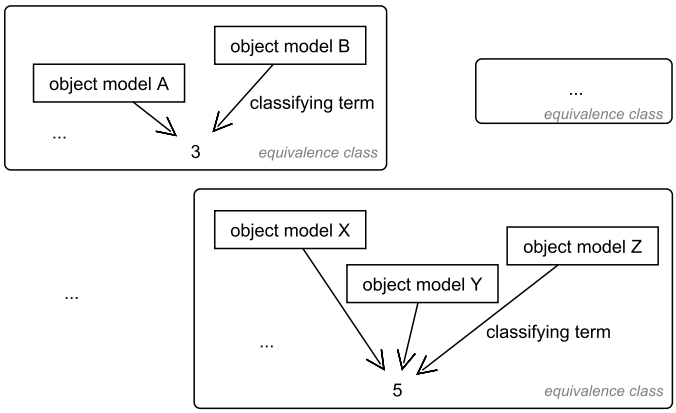


Fig. 3. Object model equivalence classes w.r.t. a classifying term.

Example: As a first simple case, a classifying term can specify the number of objects in a class. E.g., under a configuration requiring at least 2 and at most 4 Person objects, the classifying term `Person.allInstances()->size()` would yield three object models with 2, 3, and 4 Person objects, respectively.

Example: Let us continue the Parenthood example and configuration with exactly three Person objects and two Parenthood links from above. In order to prevent that the two rightmost object models from Fig. 2 are presented as different object models, the developer can employ the following classifying term.

```
Person.allInstances()->select(p |
  Person.allInstances()->exists(c,gc |
    p.child->includes(c) and
    c.child->includes(gc)))->size()
```

This term counts the number of `Person` objects that

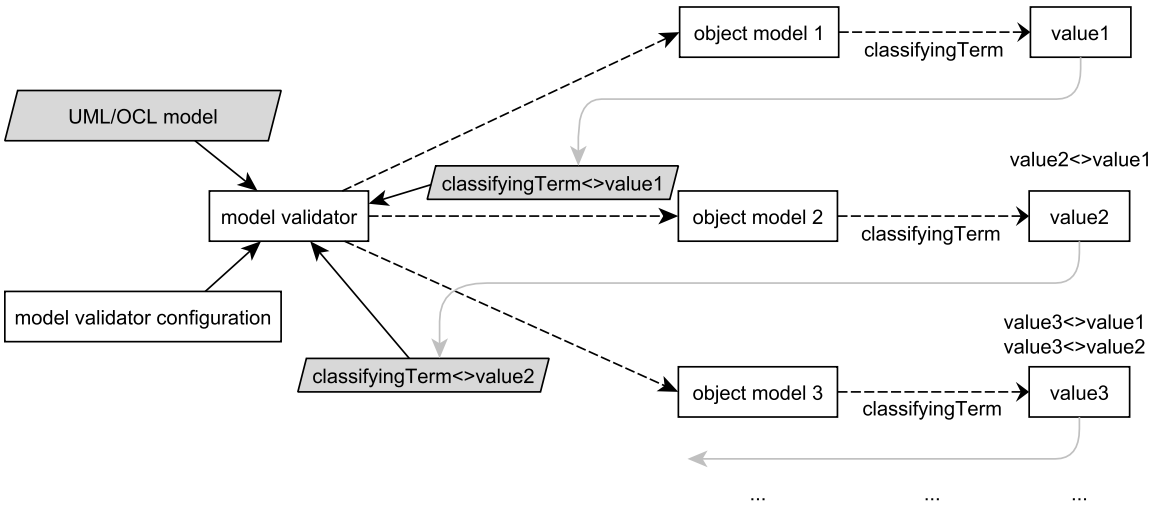


Fig. 4. Interplay between model validator and classifying term.

possess a child and a grandchild. The term rates the two rightmost object models from Fig. 2 with the same value `1`, and thus only one object model would be chosen from the corresponding equivalence class. The term rates the leftmost object model from Fig. 2 with the value `0`.

Classifying term handling: The USE model validator and a classifying term play together as depicted in Fig. 4: as an initial step, a first object model is constructed; then the value `value1` of the classifying term in the first object model is stored; afterwards, a constraint is added to the validation process, namely the constraint `classifyingTerm<>value1`; employing this constraint, a second object model is computed; the value `value2` of the classifying term in the second object model is stored, and a further constraint is added to the validation process `classifyingTerm<>value2`; the general rule is that when computing the object model N+1, the values `value1`, ..., `valueN` of the classifying term in the previous object models are used to distinguish the newly computed object model from the already found ones; these steps are repeated until no new object model is found. In our approach, classes, associations, attributes and data types must be populated with elements specified by finite sets, and thus only a finite number of object models exists.

Example: We now consider a more practical classifying term that generates structurally different object models. The configuration requires that between 1 and 3 `Person` objects and between 1 and 3 `Parenthood` links exist. The classifying term uses the boolean properties `wGp` (with grandparent), `w2c` (with 2 children) and `w2p` (with 2 parents).

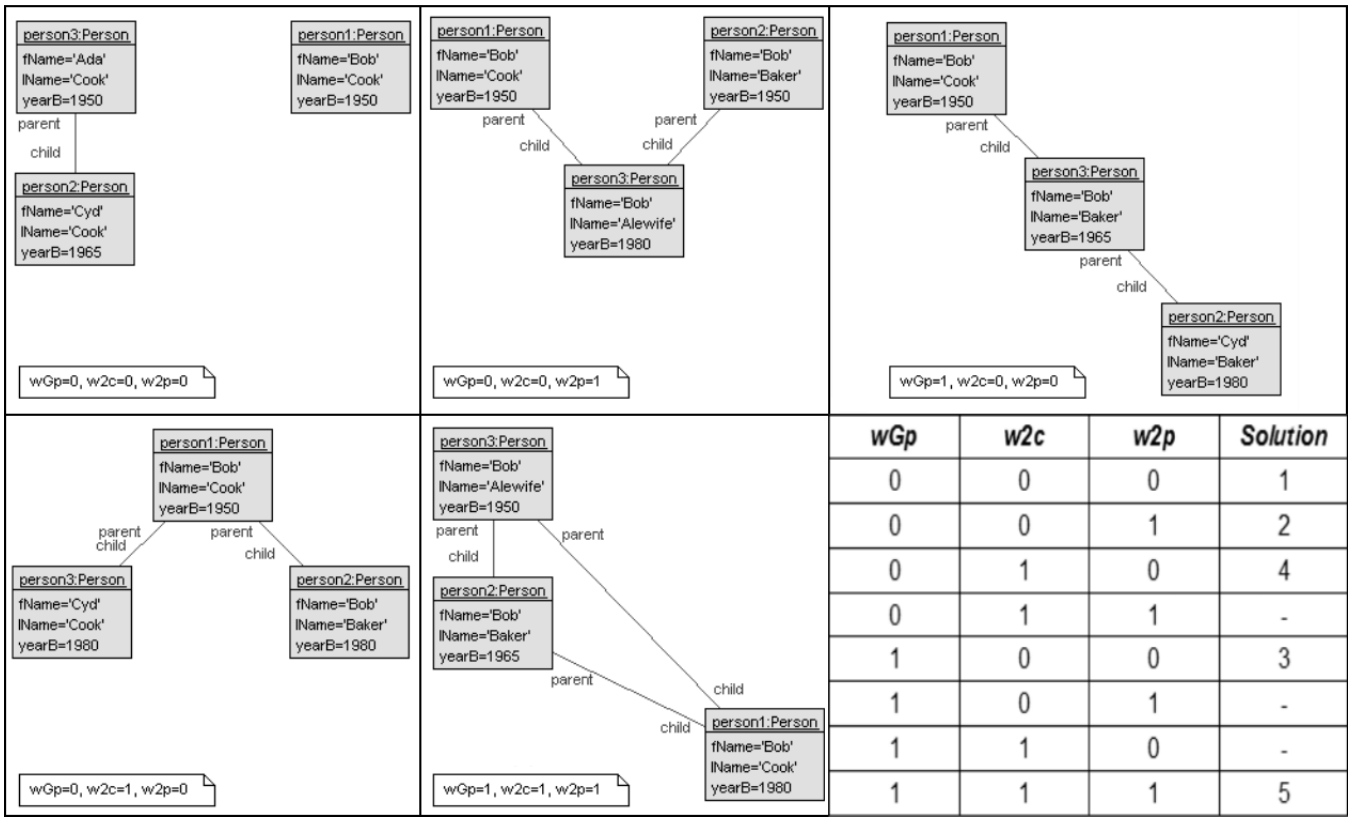
```
let P=Person.allInstances in
let wGp=P->exists(g,p,c |
            g.child->includes(p) and
            p.child->includes(c)) in
let w2c=P->exists(p | p.child->size>=2) in
let w2p=P->exists(p | p.parent->size>=2) in
if wGp then 1 else 0 endif +
if w2c then 2 else 0 endif +
if w2p then 4 else 0 endif
```

In order to obtain as many combinations as possible, the three boolean properties are considered as bits in a three-bit integer representation. The classifying term encodes this representation. The resulting object models are shown in Fig. 5. The objects models show different structural characteristics and are presented in the order in which the model validator finds them. Please note that from the possible 8 combinations of the basic boolean properties only 5 options are considered. This is primarily due to the stated configuration (1 to 3 objects, 1 to 3 links). For example, the option (wGp=0,w2c=1,w2p=1) cannot be reached with at most 3 objects, because combining w2c=1 and w2p=1 would lead to solution 5 in which wGp=1 must hold; the option (wGp=0,w2c=1,w2p=1) can be reached however by increasing in the configuration the number of objects to 4 (resulting in, e.g., p1 with children {p2,p3} and p3 with parents {p1,p4}).

As mentioned above, employing *one* classifying term of type *Integer* is one option. In general, more than one classifying term may employed. Each term is allowed to be of type `Integer` or `Boolean`. Thus the same collection of object models as in Fig. 5 may also be achieved by specifying three `Boolean` terms.

```
[ wGp ] Person.allInstances->exists(g,p,c |
          g.child->includes(p) and
          p.child->includes(c))
[ w2c ] Person.allInstances->exists(p |
          p.child->size>=2)
[ w2p ] Person.allInstances->exists(p |
          p.parent->size>=2)
```

The example demonstrates two new aspects of classifying terms. First, it is valuable to use multiple classifying terms in one validation process. And second, with multiple terms allowed, apart from integer expressions also boolean expressions can be used, which on their own only allow for at most two results. Whereas with n boolean classifying terms up to 2^n



wGp	w2c	w2p	Solution
0	0	0	1
0	0	1	2
0	1	0	4
0	1	1	-
1	0	0	3
1	0	1	-
1	1	0	-
1	1	1	5

Fig. 5. Structurally different objects models constructed by a classifying term.

possible solutions could be found. Consequently, the definition of classifying terms is extended to allow for these features.

In order to find successively new object models for a given class model plus classifying terms, the values of the classifying terms are stored for each solution. Using the classifying terms and these values, constraints are created and given to the solver along with the class model during the validation process. Informally, the constraint schema reads: *There exists no previous object model, in which the evaluation of all classifying terms in the object model currently under construction equals the stored values of the previous object models.* This statement can be formally represented as:

$$\neg \bigvee_{om \in \text{PreviousObjectModels}} \bigwedge_{ct \in \text{ClassifyingTerms}} ct = ct_{[om]}$$

ct is a classifying term and $ct_{[om]}$ refers to the stored value of the specific classifying term in the previous object model om. With this formula, the example can be realized with three distinct classifying terms and the overhead in form of the binary addition disappears, providing a more efficient solution. All described features have been implemented in the USE model validator and are available for download[1].

Advantages of classifying terms: Classifying terms can be employed for exploring the class model in order to see few diverse object models instead of many similar ones. The focus of exploration is determined by the modeler through the terms. By inspecting the constructed object models and checking their properties, the modeler gains insight into the characteristics of the class model including the OCL constraints and makes them alive. Using boolean classifying terms, one can draw conclusions which model properties (expressed as classifying terms) are allowed simultaneously in an object model (see the Table in the bottom right of Fig. 5). Thus one can analyze dependencies between requirements similar to invariant independence [6] which checks whether a given invariant is a logical consequence from other invariants. Classifying terms can employ all OCL constructs (e.g., logical connectives and collection operations as `forAll`, `collect`, `closure` or `size`) supported by the transformation into relational logic and allow to express quite general properties. They can be used to generate test cases in form of object models based on the idea of building equivalence classes.

[1]http://sourceforge.net/projects/useocl/ (USE and ModelValidator plugin)

III. Using Classifying Terms in the Context of Tracts

A. Building Tract Test Suites with Classifying Terms

Tracts: Tracts were introduced in [3] as a specification and black-box testing mechanism for model transformations. They are a particular kind of *model transformation contracts* [1, 7] especially well suited for specifying model transformations in a modular and tractable manner. Tracts provide modular pieces

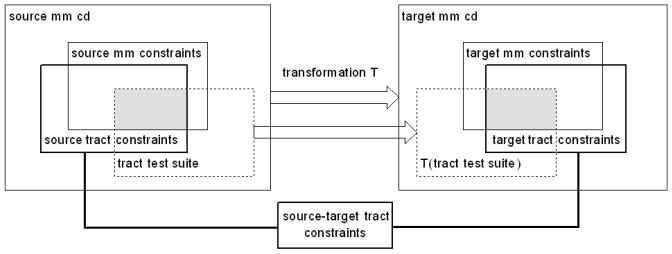


Fig. 6. Building blocks of a tract as in [3].

of specification, each one focusing on a particular transformation scenario. Thus each model transformation can be specified by means of a set of Tracts, each one covering a specific use case—which is defined in terms of particular input and output models and how they should be related by the transformation. In this way, Tracts permit partitioning the full input space of the transformation into smaller, more focused behavioral units, and to define specific tests for them. Commonly, what developers are expected to do with Tracts is to identify the scenarios of interest (each one defined by one Tract) and check whether the transformation behaves as expected in these scenarios. Tracts also count on tool support for checking, in a black-box manner, that a given implementation behaves as expected—i.e., it respects the Tracts constraints [8].

Fig. 6 depicts the main components of the Tracts approach: the source and target metamodels, the transformation T under test, and the transformation contract, which consists of a Tract *test suite* and a set of Tract constraints. In total, five different kinds of constraints are present: the source and target models are restricted by general constraints added to the language definition, and the Tract imposes additional *source*, *target*, and *source-target* Tract constraints for a given transformation. These constraints serve as "contracts" (in the sense of contract-based design [9]) for the transformation in some particular scenarios, and are expressed by means of OCL invariants. They provide the *specification* of the transformation.

If we assume a source model m being an element of the test suite and satisfying the source metamodel and the source Tract constraints given, the Tract essentially requires the result $T(m)$ of applying transformation T to satisfy the target metamodel and the target Tract constraints, and the tuple $< m, T(m) >$ to satisfy the source-target Tract constraints.

Example: In order to illustrate Tracts, consider a simple model transformation called `BiBTex2DocBook` that converts the information about proceedings of conferences (in `BibTeX` format) into the corresponding information encoded in `DocBook` format[2]. The source and target metamodels that we use for the transformation are shown in Fig. 7. Seven constraint names are also shown in the figure. These constraints are in charge of specifying statements on the source models (e.g., proceedings should have at least one paper; persons should have unique names); and on the target

[2]http://docbook.org/

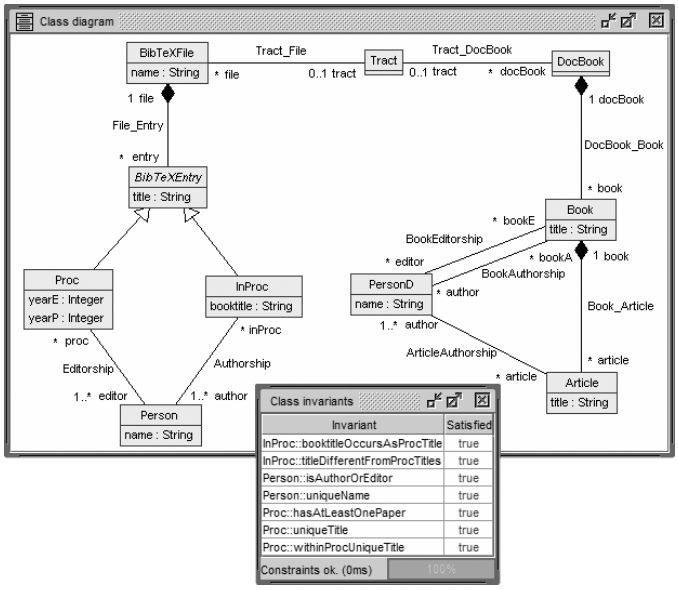


Fig. 7. Source and target metamodels.

models (e.g., a book should have either an editor or an author, but not both). The constraints for the source are shown below.

```
context Person inv isAuthorOrEditor:
  inProc->size() + proc->size() > 0

context InProc inv booktitleOccursAsProcTitle:
  Proc.allInstances->exists(prc |
    prc.title=booktitle)

context Person inv uniqueName:
  Person.allInstances->isUnique(name)

context Proc inv hasAtLeastOnePaper:
  InProc.allInstances->exists(pap |
    pap.booktitle=title)

context Proc inv uniqueTitle:
  Proc.allInstances->isUnique(title)

context Proc inv withinProcUniqueTitle:
  InProc.allInstances->select(pap |
    pap.booktitle=title)->forAll(p1,p2 |
      p1<>p2 implies p1.title<>p2.title)

context InProc inv titleDifferentFromPrcTitle:
  Proc.allInstances->forAll(p| p.title<>title)
```

In addition to constraints on the source and target models, tracts impose conditions on their relationship—as they are expected to be implemented by the transformation's execution. In this case, the `Tract` class serves to define the source-target constraints for the exemplar tract that we use (although several tracts are normally defined for a transformation, each one focusing on specific aspects or use-cases of the transformation, for simplicity we will consider only one tract here). The following conditions are part of the source-target constraints of the tract:

```
context t:Tract inv sameSizes:
  t.file->size() = t.docBook->size() and
  t.file->forAll( f | t.docBook->exists( db |
```

```
    f.entry->size() = db.book->size() ) )

context prc:Proc inv sameBooks:
  Book.allInstances->one( bk |
    prc.title = bk.title and
    prc.editor->forAll(pE | bk.editor->one(bE|
       pE.name = bE.name )))

context pap:InProc inv sameChaptersInBooks:
  Article.allInstances->one( art |
    pap.title = art.title and
    pap.booktitle = art.book.title and
    pap.author->forAll(aP |
      art.author->one(aA | aP.name=aA.name)) )
```

Tract Test Suites: In addition to the source, target and source-target tract constraints, *test suites* play an essential role in Tracts. Test suite models are pre-defined input sets of different sorts aimed to exercise the transformation. Being able to select particular patterns of source models (the ones defined for a tract test suite) offers a fine-grained mechanism for specifying the behaviour of the transformation, and allows the model transformation tester to concentrate on specific behaviours of the tract. Note that test suites may not only be positive test models, satisfying the source constraints, but also negative test models, used to know how the transformation behaves with them.

Problem: So far, the generation of test suites for tracts has been achieved using the ASSL language (A Snapshot Sequence Language) [10], which was developed to generate object diagrams for a given class diagram in a flexible way. ASSL is basically an imperative programming language with features for randomly choosing attribute values or association ends. Although quite powerful, this approach to generate source models for testing purposes presents some limitations. In particular, it makes difficult to prove some of the properties that any test suite should exhibit, such as completeness (are all possible sorts of input models covered?) and correctness (are all generated models valid and correct?). In general, analysing the coverage of the test suite w.r.t. the given tract is far from being a trivial task.

Solution: In this context, classifying terms can be of great help. They permit guiding the construction process of the test suites using *equivalence classes* that determine the sorts of input models of the tract. The process to build the test suite is then straightforward. We begin by identifying the *sorts* of models that we would like to be included in the test suite. Each sort is then specified by a classifying term, that represents the equivalence class with all models that are *equivalent* according to that class, i.e., which belong to the same sort. Once the classifying terms are defined for a Tract, the USE tool generates one representative model for each equivalence class. These *canonical* models constitute the test suite of the tract.

Example: For example, suppose that we want to concentrate on different characteristics of the input models of the `BibTex2DocBook` transformation. First, proceedings have two dates: the year in which the conference event was held (`yearE`) and the year in which the proceedings were published (`yearP`). We want to have input models in which these two dates coincide in all proceedings, and other input models with different conference event and publication years. Second, we want to have some sample input models in which two editors of proceedings invite the other to have a paper there; respectively, we also want to have input models in which this "manus-manum-lavat" situation does not happen. Finally, we want to have some source models with proceedings edited by one of the authors of the papers in the proceedings, and other input models with no "self-edited" proceedings.

Producing test suite models to cover all these circumstances by an imperative approach or by ASSL is normally tedious and error prone. However, the use of classifying terms greatly simplifies this task. It is enough to give three Boolean terms to the model validator, each one defining the classifying term that specifies the characteristic we want to identify in the model. In this case, these Boolean terms are the ones shown below.

```
[ yearE_EQ_yearP ]
  Proc.allInstances->forAll(yearE=yearP)

[ noManusManumLavat ]
  not Person.allInstances->exists(p1,p2 |
    p1<>p2 and p1.proc->exists(prc1 |
      p2.proc->exists(prc2 | prc1<>prc2 and
        InProc.allInstances->
          select(booktitle=prc1.title)->
            exists(pap2 |
              pap2.author->includes(p2) and
        InProc.allInstances->
          select(booktitle=prc2.title)->
            exists(pap1 |
              pap1.author->includes(p1))))))

[ noSelfEditedPaper ]
  not Proc.allInstances->exists(prc |
    InProc.allInstances->exists(pap |
      pap.booktitle=prc.title and
      prc.editor->
        intersection(pap.author)->notEmpty))
```

Using the specifications of these classifying terms, the model validator finds 8 solutions, which are shown in Fig. 8 in the order the model validator finds them. For each solution the value of the three properties (`yearE_EQ_yearP`, `noManusManumLavat`, `noSelfEditedPaper`) is indicated in the figure with integer values $(0, 1)$, indicating whether the particular solution fulfills the condition (1) or not (0).

In summary, we have been able to define a set of 8 equivalence classes that characterize the sorts of input models we are interested in, and have the model validator find representative (i.e., canonical) models for each class. In this way we make sure the models that constitute the tract test suite cover all cases of interest.

B. Further Analysis of Model Transformations

Due to the way in which classifying terms can be specified (by means of Boolean terms) for building the tract test suites models, they define a set of equivalence classes that constitute a (complete and disjoint) partition of the input model

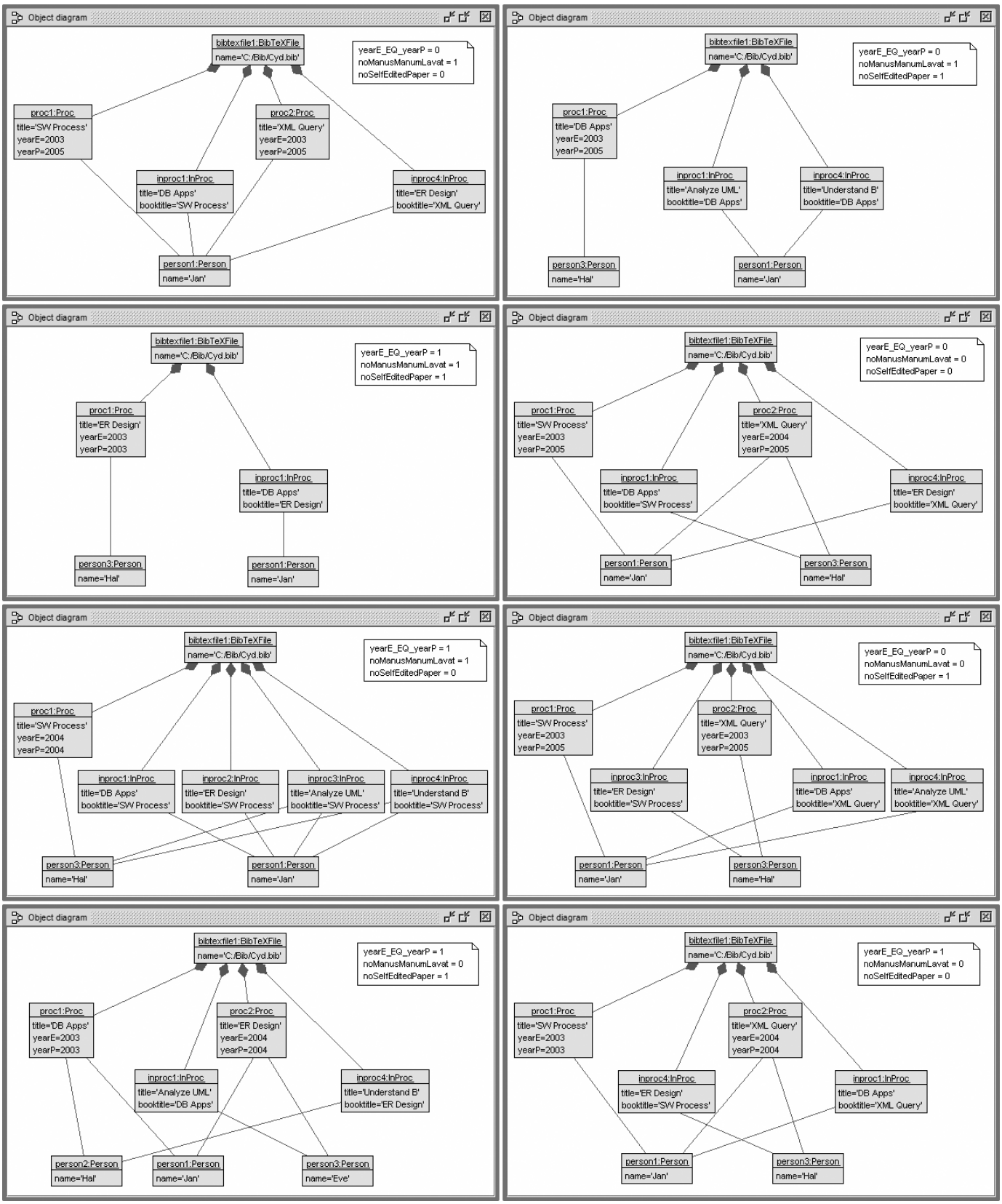


Fig. 8. The eight solutions found by the model validator.

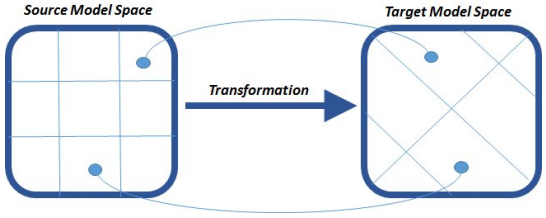


Fig. 9. Classifying terms for defining partitions of source and target spaces.

Source		Target
[0,0,0]	→	[0,0,1]
[0,0,1]	→	[0,0,1]
[0,1,0]	→	[0,0,0]
[0,1,1]	→	[0,0,0]
[1,0,0]	→	[1,0,0]
[1,0,1]	→	[1,0,1]
[1,1,0]	→	[1,0,1]
[1,1,1]	→	[1,0,0]

TABLE I
MAPPING EQUIVALENCE CLASSES.

space of the transformation. This is useful to select sample input models of different sorts (one per equivalence class), making sure that (a) we do not miss any representative model from any sort of model of interest (completeness), and (b) no two sample models are of the same kind (disjointness), as pictured in Fig. 9.

But we can also use the idea of partitioning a model space with the target model space, and characterize the sorts of target models which are of certain interest to the modeler (or to the model transformation tester). The equivalence classes defined by the target classifying terms are very useful for checking several properties of the transformation. For example, we could check that:

- All sorts of target models of interest are produced by the transformation—i.e., *full coverage* of certain parts the target model space.
- No target models of certain forms (sorts) are produced because they would be invalid target models—i.e., the transformation produces *no junk*.
- No target models of certain sorts are mapped to the same sort of target model when they shouldn't—i.e., the transformation introduces *no confusion* when it shouldn't (two models are not mapped to equal target sorts unless they belong to the same source sort).

Example: To illustrate this, let us go back to the `BibTeX2DocBook` transformation, where we can identify some sorts of models of interest in the target space.

For example, we can be interested in a property that was also of relevance in the source target space, such as *self edited papers* (i.e., whether the editor of a book is also the author of one of the chapters). We can also be interested in *normal* books, i.e., those which are not composition of papers selected by an editor, but instead all chapters are written by the same person, the book author. Finally, edited books in which no author writes more than one paper could be of interest too.

In order to specify these properties and define the appropriate equivalence classes we just need to write the corresponding classifying terms:

```
[ noSelfEditedPaper ]
  not Book.allInstances->exists(b |
    b.editor->intersection(
      b.article.author)->notEmpty() )

[ onlyNormalBooks ]
  Book.allInstances->forAll(b |
    b.editor->isEmpty() and
    b.article->forAll(a | a.author=b.author))

[ noRepeatedAuthors ]
  Book.allInstances->forAll(b |
    b.author->forAll(a |
    a.article->select(book=b)->size()=1 ) )
```

These three boolean classifying terms produce 8 equivalence classes ($8 = 2^3$) in the target model space. It is now a matter of determining the expected behaviour of the transformation with the input models from the source equivalence classes. In this respect, there are properties that should be preserved (e.g., `noSelfEditedPaper`) and others that cannot happen (e.g., given that proceedings must have at least one editor, no normal book can be generated by the transformation).

In this respect, the model validator can also be very useful to find counterexamples for situations that in principle should not happen, but that are permitted by our specification because the classifying terms are not properly defined.

With the set of equivalence classes in the source and target model spaces, we can execute the model transformation on the test suite and check whether the output models belong to the appropriate equivalence classes in the target model space.

In order to prove that, it is a matter of analysing the behaviour of the model transformation with the representative models of each source equivalence class. In this case, the mapping done by transformation for the 8 representative models of the equivalence classes (which are shown in Fig. 8) is as described by Table I.

In the table, each equivalence class is represented by a tuple $[x_1, x_2, x_3]$ where $x_i \in \{0, 1\}$ indicates if the model satisfies condition i of the corresponding classifying term. Thus, in the source model space $[1, 1, 1]$ means that the model satisfies `noSelfEditedPaper`, `noManusManumLavat` and `yearE_EQ_yearP`, while in the target model space the tuple $[1, 1, 1]$ corresponds to a model that satisfies conditions `noSelfEditedPaper`, `onlyNormalBooks` and `noRepeatedAuthors` (in this order). In this way we can check how in effect no normal books are produced. We can also see that with these input models, all the rest of the equivalence classes that we have defined for the target space have been reached.

C. Selecting more than One Sample per Classifying Term

So far, we have been able to check that indeed the behaviour of the transformation is as expected for the selected sample models. However, this does not prove that the transformation

will always work. What would have happened if the model validator would have selected other representative models for the equivalence classes?

This may happen, for instance, when the equivalence classes are not defined at the appropriate level of granularity (either in the source or target model spaces). In this case, two input models of the same source equivalence class would be transformed into two different target equivalence classes.

This is why it would be interesting to ask the model validator to produce more than one model for each equivalence class. There is another good reason for that: we know that not all sorts of input models have the same likelihood of happening in the source model space. Thus, we can select more sample models for those equivalence classes that we think are more frequent. In this way we can exercise the model transformation in a more focused manner, and produce a richer test suite for the tract (and hence for the transformation).

In order to ask the model validator to produce more than one object model for each equivalence class, one could specify additional 'second-level classifying terms' that only apply to non-empty 'first-level' equivalence classes. For example, a second-level classifying term for the source model of the `BibTeX2DocBook` example could be:

```
[ exactlyOnePaperInProc ]
  Proc.allInstances->forAll(prc |
    InProc.allInstances->select(pap |
      pap.booktitle=prc.title)->size()=1)
```

This term could produce for the second equivalence class in Fig. 8 (in which the proceedings object has two papers) another representative with only one paper within a proceedings. Working out the details for this sketch is left for future work. However, in this way one could declaratively select a set of input models that will constitute the test suite of the tract, deciding not only the sorts of models that we are interested in, but also how many different sample models of each sort we want.

IV. Related Work

With respect to the contribution of this paper, we first present related approaches which are dedicated to generate object models in a (semi-)automated manner, and second we discuss related work considering approaches for testing and verifying model transformations.

A. Generating Object Models

The USE model validator, used in this work, is based on the transformation of UML and OCL into relational logic [11]. Many approaches exist to generate object models from class models using different languages and tools. Another approach within the same tool, USE, is the Automatic Specification Snapshot Language (ASSL) [10], which uses an iterative method to generate an object model from a given specification.

Further approaches rely on different technological cornerstones like logic programming and constraint solving [12], relational logic and Alloy [13], term rewriting with Maude [14] or graph grammars [15]. In contrast to the tool used in this work, these approaches either do not support full OCL (e.g., higher-order associations [13] or recursive operation definitions [12] are not supported) or do not facilitate full OCL syntax checks [14]. Also, the feature to automatically scroll through several valid object models from one verification task is not possible in all of the above approaches.

(Semi)-automatic proving approaches for UML class properties have been put forward on the basis of description logics [16], on the basis of relational logic and pure Alloy [13] using a subset of OCL, and in [17] focusing on model inconsistencies by employing Kodkod. A classification of model checkers with respect to verification tasks can be found in [18].

The idea of classifying terms has similarities to the analysis of invariant independence [6]. The goal is to find invariants that are fully covered by means of other invariants or class model inherent constraints (e.g. multiplicities). The goal can be achieved using boolean classifying terms, resulting in detailed information about which invariants can be satisfied independently of others.

B. Testing and Verifying Model Transformations

In the field of Model-Driven Engineering, testing and analysis of model transformations has been subject to investigations (see, for example, [19, 20]). Regarding dynamic approaches, for which the model transformation execution is needed and therefore input models, the authors in [21] and [22] present their contribution for debugging model transformations. Also, the work in [23] analyse the execution traces between the source and target models in order to find errors, and in [24] a white-box test model generation approach for testing the transformations is proposed. In this context, Tracts [25] are a complementary approach that establishes contracts between the source and target metamodels which define the transformation specification.

In addition to Tracts, other static approaches have been proposed such as [26] that allows the specification of contracts in a visual manner, and [27] that looks at the differences between the actual output model generated by the transformation and the expected output model. The first one also relies on OCL to give the user full expressiveness while the second one needs the developer to provide output models – which is not always a feasible task, and if feasible, it might require a lot of time and effort.

A test-driven method [28] is also proposed in the field of model transformation for which the model transformation implementation itself is annotated by the transformation developer removing the need of an independent specification description. A solution for the QVTo language [29] is available and presented in [30]. Although achieving its goal, making the specification of the transformation implementation-dependent prevents the separation of concerns, which is even more serious in the field of MDE as there is no dedicated standard transformation language.

Finally, equivalence partitioning [2] is a software testing technique that assumes that the inputs of the program can be divided into mutually exclusive classes according to the

behavior of the program on those inputs and, in some cases, on the outputs. In this regard, the work in [1] proposed to pick a set of relevant properties for the input models, define ranges of values for each property and check that there is at least one instance of each property that has one value in each range. Nevertheless, this proposal is less expressive than classifying terms as they do not consider the use of OCL, less flexible and lacks full automation. In [31], a mechanism for generating test cases by analysing the OCL expressions in the source metamodel in order to partition the input model space was presented. This is a systematic approach similar to ours, but focusing on the original source model constraints. Our proposal allows the developer partitioning the source (and target) model space independently from these constraints, in a more flexible manner.

V. Conclusions

This contribution has introduced classifying terms, an instrument for exploring object models in the context of a UML class model and accompanying OCL constraints. Classifying terms allow the developer to construct relevant test cases in form of object models in a goal-oriented way. Classifying terms determine equivalence classes of test cases, selection of representatives and exploration of model properties. Their usefulness has been demonstrated by generating input test models for model transformations.

Our work can be continued in various directions. The translation to relational logic can be improved and extended, for example, by considering further collection kinds. The current user interface for classifying terms is minimal, names could be given to the terms, and these names together with the values could be indicated in the resulting object models. The restriction, that only integer and boolean terms are used, can be weakened, at least enumerations do not present any problem. It would be interesting to consider more than one equivalence class representative by distinguishing between first and second level classifying terms, where second level terms are only applied for non-empty first level equivalence classes. Larger case studies should give more feedback on the features and scalability of the approach. Last but not least, particular tool support for model transformations with different options for source and target is needed.

Acknowledgment

This work was partially funded by the German Research Foundation (DFG) under grant GO 454/19-1, by the Spanish Research Projects TIN2011-23795 and TIN2014-52034-R and by Universidad de Malaga (Campus de Excelencia Internacional Andalucia Tech).

References

[1] B. Baudry, T. Dinh-Trong, J. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon, "Model transformation testing challenges," in *ECMDA WS. on Integration of MDD and Model Driven Testing*, 2006.

[2] I. Burnstein, *Practical Software Testing*. Springer-Verlag, 2003.

[3] M. Gogolla and A. Vallecillo, "*Tract*able model transformation testing," in *Proc. of ECMFA'11*, ser. LNCS, no. 6698. Springer, 2011, pp. 221–236.

[4] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[5] E. Torlak and D. Jackson, "Kodkod: A Relational Model Finder," in *Proc. of TACAS'07*, 2007, pp. LNCS 4424, 632–647.

[6] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, independence and consequences in UML and OCL models," in *Proc. of TAP'09*, ser. LNCS, vol. 5668. Springer, 2009, pp. 90–104.

[7] 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.

[8] 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.

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

[10] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL Models in USE by Automatic Snapshot Generation," *Software and Systems Modeling*, vol. 4, no. 4, pp. 386–398, 2005.

[11] M. Kuhlmann and M. Gogolla, "From UML and OCL to relational logic and back," in *Model Driven Engineering Languages and Systems*, ser. LNCS, vol. 7590. Springer, 2012, pp. 415–431.

[12] J. Cabot, R. Clarisó, and D. Riera, "UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models using Constraint Programming," in *ASE 2007*. ACM, 2007, pp. 547–548.

[13] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "On Challenges of Model Transformation from UML to Alloy," *Software and System Modeling*, vol. 9, no. 1, pp. 69–86, 2010.

[14] M. Roldán and F. Durán, "Dynamic Validation of OCL Constraints with mOdCL," *ECEASST*, vol. 44, 2011.

[15] K. Ehrig, J. M. Küster, and G. Taentzer, "Generating instance models from meta models," *SoSyM*, vol. 8, pp. 479–500, 2009.

[16] A. Queralt, A. Artale, D. Calvanese, and E. Teniente, "OCL-Lite: Finite reasoning on UML/OCL conceptual schemas," *Data Knowl. Eng.*, vol. 73, pp. 1–22, 2012.

[17] R. V. D. Straeten, J. P. Puissant, and T. Mens, "Assessing the Kodkod Model Finder for Resolving Model Inconsistencies," in *ECMFA*, ser. LNCS, vol. 6698. Springer, 2011, pp. 69–84.

[18] S. Gabmeyer, P. Brosch, and M. Seidl, "A Classification of Model Checking-Based Verification Approaches for Software Models," 2013, Proc. of the 1st VOLT Workshop.

[19] *Proc. of the AMT WS.*, ser. CEUR WS. Proc., vol. 1277, 2014.

[20] *Proc. of the VOLT WS.*, ser. CEUR WS. Proc., vol. 1325, 2014.

[21] 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.

[22] 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*, 2009.

[23] V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser, "Traceability mechanism for error localization in model transformation," in *Proc. of ICSOFT'09*, 2009.

[24] 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.

[25] 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)*. Springer, 2012.

[26] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Automated verification of model transformations based on visual contracts," *Autom. Softw. Eng.*, vol. 20, no. 1, pp. 5–46, 2013.

[27] 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, no. 6981. Springer, 2011, pp. 395–409.

[28] P. Giner and V. Pelechano, "Test-driven development of model transformations," in *Proc. of MODELS'09*, ser. LNCS. Springer, 2009, vol. 5795, pp. 748–752.

[29] OMG, *Meta Object Facility (MOF) 2.0 Query/View/Transformation. Version 1.1*, Object Management Group, 2011.

[30] A. Ciancone, A. Filieri, and R. Mirandola, "MANTra: Towards model transformation testing," in *Proc. of QUATIC'10*. IEEE, 2010, pp. 97–105.

[31] C. A. González and J. Cabot, "Test Data Generation for Model Transformations Combining Partition and Constraint Analysis," in *Proc. of ICMT'14*, ser. LNCS, vol. 8568. Springer, 2014, pp. 25–41.