International Journal on Software and Systems Modeling (SoSyM) manuscript No.
(will be inserted by the editor)

Testing Models and Model Transformations using Classifying Terms

Frank Hilken · Martin Gogolla · Loli Burgueño · Antonio Vallecillo

Received: date / Accepted: date

Abstract This paper proposes the use of Equivalence Partitioning techniques for testing Models and Model Transformations. In particular, we introduce the concept of Classifying Terms (CT), which are arbitrary OCL terms on a class model enriched with OCL constraints. CTs permit defining Equivalence classes for partitioning the source and target model spaces of the transformation, defining each class a set of equivalent models with regard to the transformation. Using these classes, a model validator tool is able to automatically construct object models for each class, which constitute relevant test cases for the transformation. We show how this approach of guiding the construction of test cases in an orderly, systematic and efficient manner can be effectively used in combination with Tracts for testing both directional and bidirectional model transformations and for analysing their behavior.

Keywords Model Transformations · Contract-based Specifications · Equivalence Partitioning

1 Introduction

Model transformations (MT) are being increasingly used in many different contexts. From simple structural migration, model queries or pattern-based code-generation, they now have to cope with complex model synthesis, behavioural analysis and stream data processing. This has led to a significant increase in their complexity and, hence, to the need of *engineering* model transformations [29].

In this context, the specification and testing of model transformations become critical tasks to ensure the correctness of their implementations. Note that correctness is not an absolute property. It needs to be checked against a contract, or specification, which determines the expected behaviour, the context in which such a behaviour needs to be guaranteed, as well as some other properties of interest. The specification states what should be done, but

F. Hilken and M. Gogolla
University of Bremen, Germany
E-mail: {fhilken,gogolla}@informatik.uni-bremen.de

L. Burguenõ and A. Vallecillo
University of Málaga, Spain
E-mail: {loli,av}@lcc.uma.es

without determining how. The problem, again, is that the specification of a model transformation can be as complex as the transformation itself. This is why modular techniques are needed for specifying and testing model transformations.

One of the problems of existing model transformation testing techniques lies in the difficulty of selecting effective test cases [4]. In this paper we explore the use of *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 [7]. 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 [25], 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. More specifically, we show how this approach of guiding the construction of test cases in an orderly, systematic and efficient manner can be effectively used in the specification and testing of both directional and bidirectional model transformations.

This paper is organized in 7 sections. After this Introduction, Section 2 introduces classifying terms, describes how they are specified, and presents the mechanism available for automatically constructing the representative object models. Then, Section 3 introduces Tracts and Section 4 describes how classifying terms can be used in the context of Tracts to implement model transformation testing. Section 5 goes a step further and also shows the use of Tracts and classifying terms to specify and test bidirectional model transformations. Finally, Section 6 relates our work to other similar approaches and Section 7 concludes and outlines some future lines of work.

2 Classifying Terms

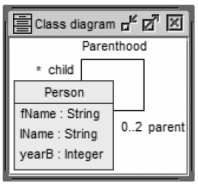
Classifying terms are an instrument to explore model properties. We discuss their underlying concepts and their implementation in the context of a tool, the UML-based Specification Environment (USE). The underlying ideas can be employed however in similar modeling tools. USE allows the modeler to describe 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.

2.1 USE Model Validator

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 [34] of Kodkod [47], (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.

2.2 The Concept of Classifying Term

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.

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

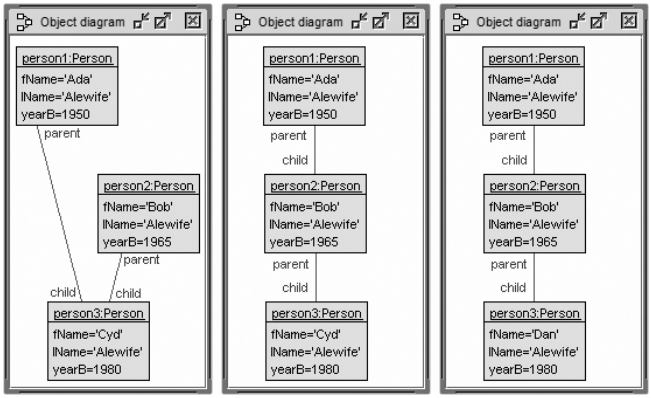


Fig. 2 Different example object models with partly isomorphic structure.

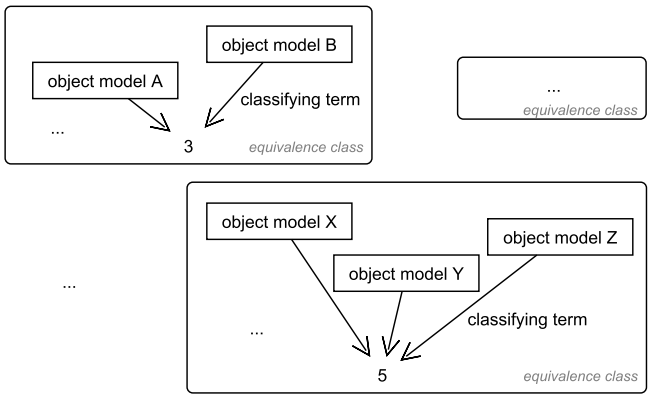


Fig. 3 Object model equivalence classes w.r.t. a classifying term.

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.

Example 1 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,

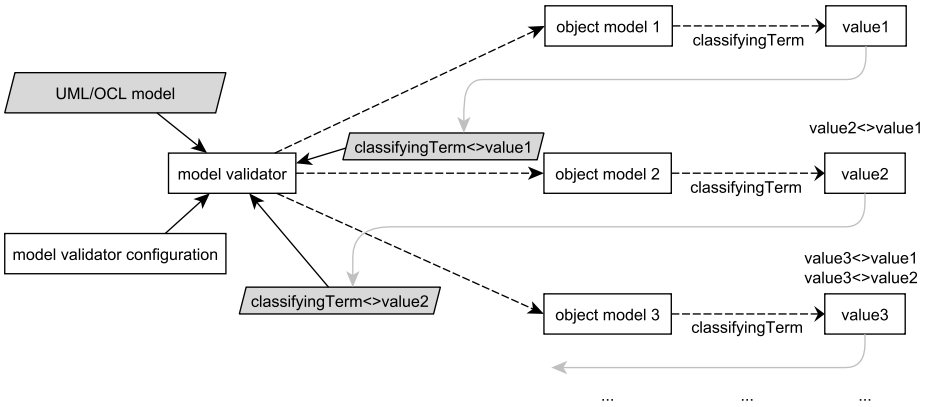


Fig. 4 Interplay between model validator and classifying term.

the classifying term `Person.allInstances()->size()` would yield three object models with 2, 3, and 4 Person objects, respectively.

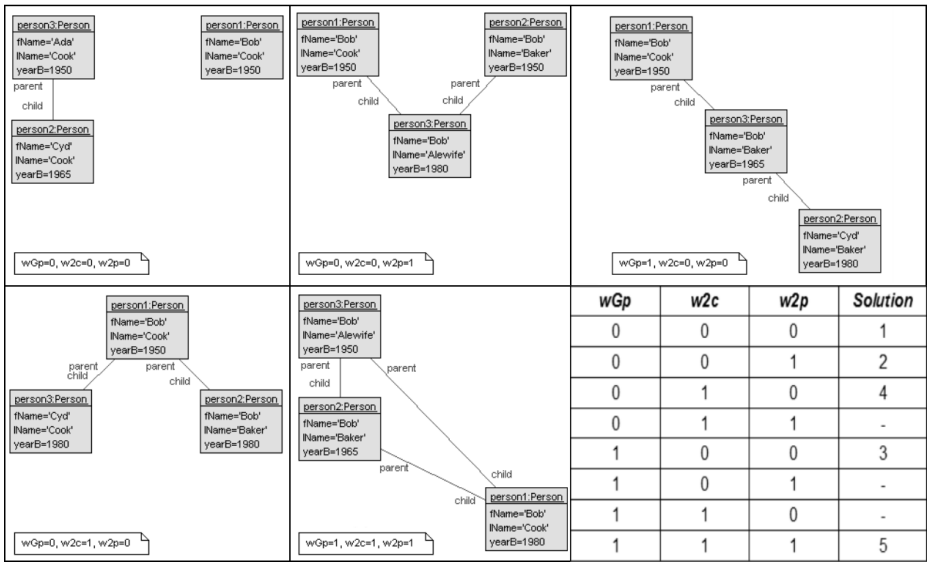
Example 2 Let us continue the Parenthood example and configuration with exactly three Person objects and two Parenthood links from above. In order to prevent that the two rightmost object models from Fig. 2 are presented as different object models, the developer can employ the following classifying term.

```
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 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`.

2.3 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.



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.

Example 3 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 be 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 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].

2.4 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 [24] 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.

Furthermore, equivalence class partionioning is well-known and the classifying terms inherit their advantages. Depending on the chosen classifying terms, the amount of generated

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

```
function nextSolution:
    In: solverInstance // constraints representing the model and bound configuration
    InOut: termValues // storage for values of classifying terms, initially empty
    Out: result // solution instance within new equivalence class
    solverInstance.CTConstraint ← genCTConstraint(termValues)
    // run SAT solver
    result ← solve(solverInstance)
    if isSatisfiable(result) then
        // record CT values for this solution
        termValues ← termValues ∪ readTermValues(result)
    end
end
```

Algorithm 1 Iteration step to generate the next solution.

system states (test cases) is significantly reduced. In addition, each test case has a higher potential of revealing defects, due to the diversity enforced by the classifying terms. Finally, models become increasingly larger and although the testing is limited by the chosen bounds, exhaustive testing of every single possible system state is not feasable.

2.5 Algorithm and Implementation

The USE model validator uses an iterative algorithm to successively generate the solution instances for the given model and classifying terms. The steps produce a sequence like the one shown in Fig. 4. After each step, the generated system state is loaded in USE and can be inspected, e.g., using the visual representations or evaluating OCL queries on it. The user can issue a command of the model validator to initiate the generation of the next solution. Alternatively, the model validator can be configured to generate the complete sequence of system states at once without user interaction.

Algorithm 1 shows the pseudo code of the iteratively invoked function *nextSolution* which outlines an iteraton step in the generation of system states for each equivalence class. The *solverInstance* contains the information from the model and the bound configuration required by Alloy/Kodkod [35]. This input is prepared in advance of the iterations, since the UML/OCL model and bound configurations are fixed for the duration of the process. Next, the variable *termValues* is a list of mappings that stores the evaluated result per classifying term for each generated system state. The function `genCTConstraint` generates the formula to exclude system states that are in any of the equivalence classes of previous solutions. The function implements the formula presented in Sect. 2.3. For the first run of the function, *termValues* is empty and thus the generated constraint results in a tautology, not restricting the generation of a system state.

Now, the *solverInstance* is complete and ready to be solved by a SAT solver using the Kodkod library [47]. The result is either of two possibilities. In the first case, the solver does not find a valid assignment and yields an *unsatisfiable*. This means, within the restrictions of the bounds and classifying terms, there is no further solution, i.e., no system state that is not within any of the previous equivalence classes. This case terminates the iterative algorithm. In the other case, the solver is able to find an assignment and yields a *satisfiable*. This implies that a new equivalence class has been found and the system state is analyzed for the unique classifying terms values, which are added to the *termValues* in preparation for the next iteration.

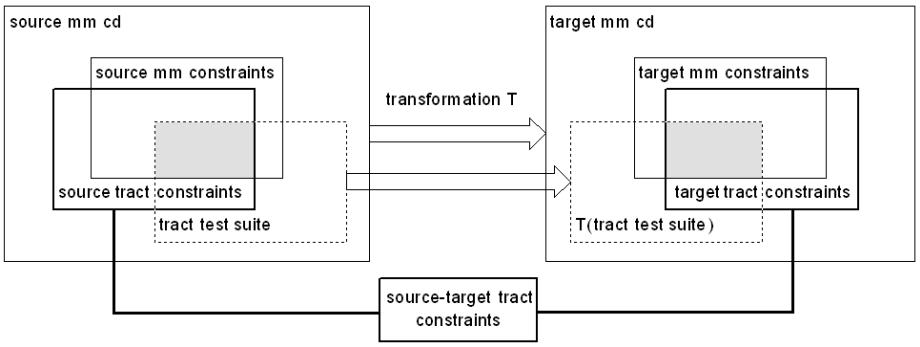


Fig. 6 Building blocks of a tract as in [25].

3 Tracts

3.1 The concept of Tract

Tracts were introduced in [25] as a specification and black-box testing mechanism for model transformations. They are a particular kind of *model transformation contracts* [4,9] especially well suited for specifying model transformations in a modular and tractable manner. Tracts provide modular pieces 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 [6].

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 [37]) 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

[2] `http://docbook.org/`

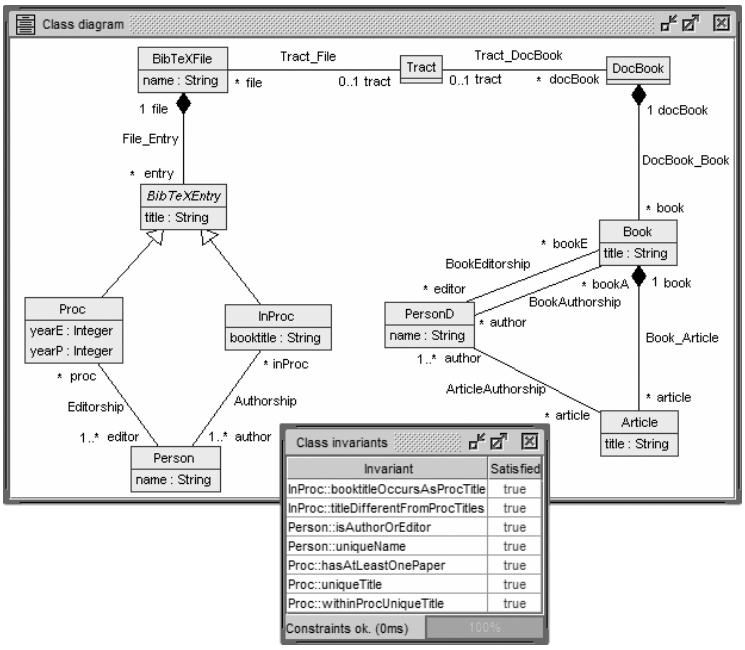


Fig. 7 Source and target metamodels.

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 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 exe-

cution. 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->selectByType(Proc)->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:
  Chapter.allInstances->one( chp |
    pap.title = chp.title and
    pap.booktitle = chp.book.title and
    pap.author->forAll(aP | chp.author->one(cA | aP.name=cA.name)) )
```

3.2 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.

So far, the generation of test suites for tracts has been achieved using the ASSL language (A Snapshot Sequence Language) [23], 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.

4 Using Classifying Terms in the Context of Tracts

4.1 Building Tract Test Suites with Classifying Terms

As mentioned above, classifying terms can be of great help in this context. 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: 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 that 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.

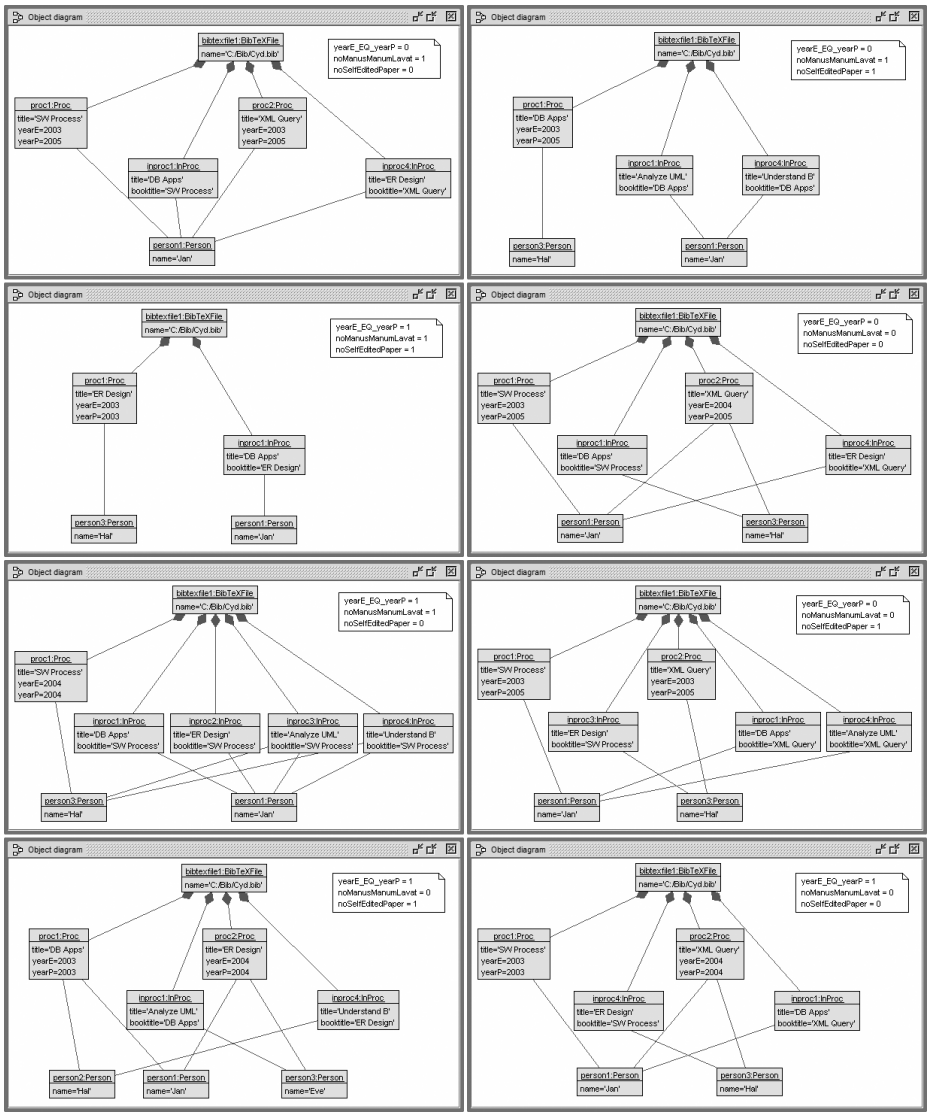


Fig. 8 The eight solutions found by the model validator.

4.2 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 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.

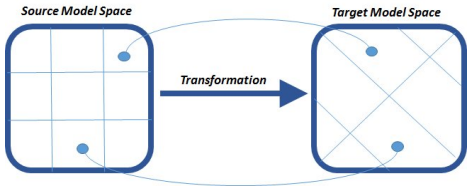


Fig. 9 Classifying terms for defining partitions of source and target spaces.

But we can also apply the idea of partitioning a model space with the target domain, characterizing 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 model space.

For instance, 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, 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.chapter.author)->notEmpty() )

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

[ noRepeatedAuthors ]
  Book.allInstances()->forAll(b |
    b.chapter->forAll( c1, c2 |
      c1 <> c2 implies c1.author->intersection(c2.author)->isEmpty()))
```

These three boolean classifying terms produce only 6 equivalence classes in the target model space, instead of the 8 ($8 = 2^3$) that could be expected. This is because self-edited papers cannot be at the same time normal books, i.e., negation of `noSelfEditedPaper` and `onlyNormalBooks` exclude each other.

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

Source		Target
$[0,0,0]$	$\longrightarrow$	$[0,0,1]$
$[0,0,1]$	$\longrightarrow$	$[0,0,1]$
$[0,1,0]$	$\longrightarrow$	$[0,0,1]$
$[0,1,1]$	$\longrightarrow$	$[0,0,0]$
$[1,0,0]$	$\longrightarrow$	$[1,0,1]$
$[1,0,1]$	$\longrightarrow$	$[1,0,1]$
$[1,1,0]$	$\longrightarrow$	$[1,0,0]$
$[1,1,1]$	$\longrightarrow$	$[1,0,1]$

Fig. 10 Mapping equivalence classes.

that should be preserved by the transformation (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 order to check that, it is a matter of analysing the behaviour of the model transformation with the representative models of each source equivalence class. Thus, 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 this case, the mapping done by transformation for the 8 representative source models of the equivalence classes (which are shown in Fig. 8) is as described by Fig. 10.

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 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).

Thus we can see how in effect no normal books have been produced when the transformation is executed on the source models. 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.

Possible misbehaviours of a model transformation detected using this approach may be due to several causes. In the first case, the equivalence classes of the transformed models in the target model space do not coincide with the expected ones. This would mean a problem in the implementation of the transformation. But it could also be the case of a wrong definition of the source or target classifying terms, which would uncover a potential mistake in the way the designer expects the transformation to work. 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 either the classifying terms or even the tracts themselves are not properly defined, as discussed in [32].

4.3 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.

5 Using Tracts and CTs for BX Testing

An important task is to test that the implementation of a *bidirectional transformation* (BX) between two metamodels conforms with the specification given by the designer and respects its behavior. In this section, we use classifying terms in combination with Tracts to systematically explore the transformation and check that the behavior of its implementation conforms with its expectations.

5.1 Bidirectional Transformations

When we think of a transformation we tend to consider a *directional* mapping between a source and a target artefact that establishes a relationship between them. But other kinds of model transformations are also gaining acceptance, in particular *bidirectional transformations*, which are responsible for checking if two (or more) models are consistent according to the relationship established by the transformation, being able to restore the consistency between them in case they are not [42,43]. A BX can be seen as two directional transformations that allow creating such links in both directions [12], subject to some properties such as correctness, hippocraticness, and sometimes history invariance [14] or undoability [43]—or their analogous laws for lenses: PUTGET, GETPUT and PUTPUT [18]. BX have many interesting applications including the synchronization of replicated data in different formats [38], presentation-oriented structured document development [33] or to implement coupled software transformations [36].

Bijective transformations constitute the simplest case of BX, where every source model is related to exactly one target model according to the relationship defined by the transformation. In this case, the overall information managed by the two metamodels is exactly the same, and there is no choice about what the transformation should do to restore consistency when one of the models changes: given a source (resp. target) model, it must return the unique target (resp. source) model which is correctly related to it.

However, bijective transformations are rare in practice because the metamodels related by a BX normally address different concerns, and hence contain different information [43]. A normal situation is when one metamodel is a projection of the other. In fact, BX initially originated in the database community to address the view-update problem, where one of the models (the view) is a strict abstraction of the other (the database). Propagating changes from the database to the view is easy, but propagating them in the other direction (e.g. the addition of a new element) may not be easy because the view does not necessarily contain all the information required to restore the consistency [15]. Nevertheless, there are some properties that need to be ensured in all cases to guarantee the soundness of the relationship defined by the transformation. This has led to the concept of *lenses* [18] with properties such as PUTGET and GETPUT for *well-behaved* lenses, or weaker versions such as PUTGETPUT = PUT [19], PUTGETPUT $\sqsubseteq$ PUT, or GETPUTGET = GET [38].

In the general case, none of the metamodels is a view of the other, and therefore both handle information which is not reflected in the other. In this case, some properties such as *correctness*, *hippocraticness* or *undoability* should be ensured so that the BX is indeed capable of synchronizing and maintaining the consistency between the models involved in a sound and reasonable manner [43, 44, 16].

5.2 Relating the Information of both Metamodels

A BX between two metamodels should be able to create one model from the other if it is missing, or to restore consistency between them if they both exist and one of them is modified. The problem with any non-bijective BX is that there are some issues that the developer has to face for which no easy solution exists. The first one is about how to deal with the information that is missing in the target metamodel, as it happens in our example case study with the years of the celebration of the event and the publication of the proceedings. Then, if a Book is created by a librarian in the target model, how does the BX propagate that change to the source? Which years are used in the corresponding source Proceedings?

Furthermore, there may be target models that do not correspond to any valid model in the source domain. This could not happen if the target is a view (and hence a refinement) of the source, but this is not our case. For instance, some books may have no editors, or may have chapters with the title of the book, something which is not permitted for proceedings. Similarly, normal books are permitted in the target models but they do not correspond to any valid kind of proceedings in the source. Then, imagine that we start with a synchronized pair of models, and the librarian decides to change a book in the target domain by making the editor become the author, or making the title of one of its chapters coincide with the book title. How does the BX behave with these target models when the consistency between the source and target models should be restored?

In general terms, the question is now how to check that the behavior of a given implementation of a BX between these two metamodels is correct, or at least conforms to what the designer expects from it. For that we need to be able to specify the expected behavior,

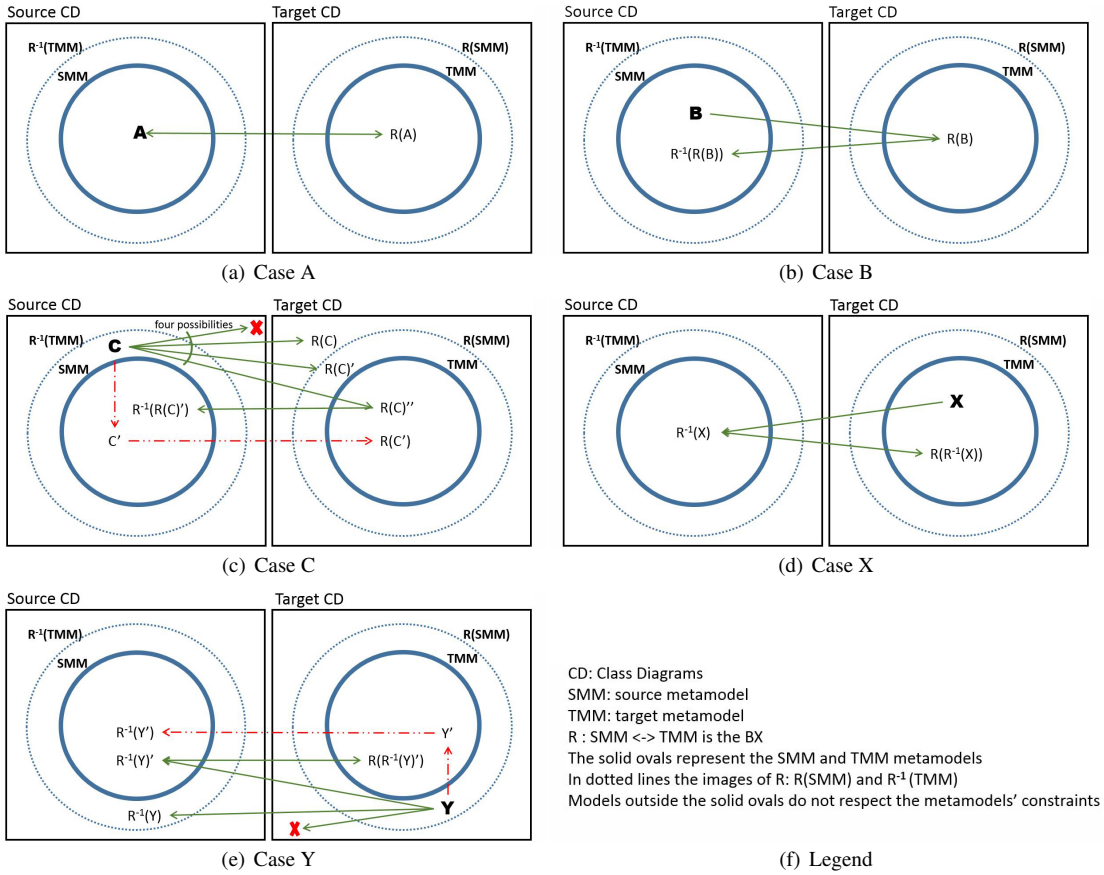


Fig. 11 Possible situations when transforming elements forward and backwards.

and then check that the implementation respects such a behavior. This is precisely the goal of this Section.

The behavior of the model transformation is clear and easy to specify for those models in which the relationship between the source and target models is a bijection. Similarly, for those models that belong to the image of the forward or backward transformation (i.e., those that handle shared information between the two metamodels), the properties of *correctness* and *hippocraticness* ensure a sound (or at least well-defined) behavior. The problem resides in those models outside the images of the BX. Correctness ensures that after the transformation, the models are consistent. Hippocraticness ensures that if they were already consistent, the transformation does nothing [43].

The diagram shown in Fig. 11 illustrates these situations, as well as the corresponding *reconciliation patterns*. Starting from the source metamodel `SMM`, some elements (`A`) are transformed in a bijective way to and from `R(A)`. Other models, such as `B`, are transformed to `R(B)` but their reverse image $\texttt{R}^{-1}\texttt{(R(B))}$ does not coincide with `B` when transformed back (note that hippocraticness ensures that the relation between $\texttt{R}^{-1}\texttt{(R(B))}$ and `R(B)` is maintained henceforth).

Finally, some elements such as C do not match all the source constraints, and therefore the behavior of the transformation cannot be guaranteed. The reason for considering these kinds of elements is that they can be produced by the transformation when reconciling elements of the target metamodel outside the image of the forward transformation (this happen for instance, with $R^{-1}(Y)$).

We have discovered that these kinds of elements (C or $R^{-1}(Y)$) can be treated by the transformation in four different ways: (a) they are ignored by the BX and not transformed; (b) they are transformed into elements (e.g., R(C)) that do not conform to the target metamodel (i.e., they violate some of the metamodel constraints or some of the Tract target constraints); (c) they are transformed into correct elements (e.g., R(C)') but outside the expected image of the forward transformation; or (d) they are transformed into elements inside the image of the forward transformation (R(C)''). From that moment on, the reconciliation pattern follows one of the situations described above for elements such as X or Y.

We have also seen that some model transformation engines exhibit a different behavior, as it happens for instance with Medini-QVT: instead of transforming model C, only the subset of that model that respects the source metamodel constraints (C') is considered by the BX and then transformed. This deviation from the expected behaviour is shown in Fig. 11 using a dotted line. Another interesting behavior that we have found is that the transformation may establish consistent relations even between elements which do not conform to the metamodel, as it happens with $R^{-1}(R(C))$ and C, once they are related. The reason is that the trace model created by the transformation is strongly used by some BX implementation engines (such as Medini-QVT or JTL), even when the related models are not fully correct.

The diagram in Fig. 11 also shows the equivalent situations when the backward transformation is considered (X playing the role of B, and Y the role of C). Note that for readability reasons we have omitted some reconciliations because they follow the same patterns as shown for other elements. This is the case for R(C') and $R^{-1}(Y')$, whose reconciling behavior is similar to that of R(C)' and $R^{-1}(Y)'$, respectively. Similarly, it could be the case that $R^{-1}(Y)'$ is reconciled either with Y (even if it falls outside the image of R) or with $R(R^{-1}(Y'))$, depending on whether the transformation engine makes use or does not make use of the trace model created when transforming Y to $R^{-1}(Y)'$.

For the BX user (and for the developer) it is very important to understand how the transformation behaves in all these circumstances. This is why we need to be able to count on specification and testing mechanisms and tools that permit representing these kinds of elements and then checking that the behavior of the implementation of the BX is as expected (and captured by the specifications). And this is where classifying terms in combination with Tracts can play a significant role.

5.3 BX Specification using Classifying Terms and Tracts

Tracts are direction-neutral, and hence they can be read and used to specify the bidirectional relationship that a BX defines between the metamodels involved in the transformation. In particular, the Metamodel shown in Fig. 7 specifies the BibTex2DocBook Tract and it can be naturally read in both directions.

Classifying terms are also very useful in this context because each of them can serve to characterize a particular property of interest in the source or target model spaces. In particular, we would like to capture and represent the situations described above (and depicted in Fig. 11) to understand how not only the transformation but also the reconciliation process works in all possible circumstances, avoiding possible surprises. This is particularly relevant

in the case of BX engines, whose behavior and semantics are normally underspecified and hence difficult to predict [43].

Example: in our case study we are interested in a property (`noSelfEditedPaper`) that is shared and relevant to the two domains and we want to check whether it is preserved by the transformation in both directions. Other classifying terms characterize elements that can only happen in one of the domains, such as `yearE_EQ_yearP` in the source and `onlyNormalBooks` in the target. The first one deals with information only available in the source domain (the years of the publication) and the second characterizes some kinds of publications which are not possible in the source domain (namely normal books, which do not have editors, but in fact are the most common kinds of book in a library). Finally, the other two classifying terms characterize properties that focus on specific aspects of the particular domain. In this case `noManusManumLavat` captures some aspects of ethical concern in the editing of proceedings and `noRepeatedAuthors` identifies books in which no author writes more than one paper. They should be preserved by the transformation in the other domain, but there they do not have any significance and therefore they can be abstracted away when making changes and selecting representative models according to the counterpart equivalence classes.

Regarding the Tract constraints, we already specified the source and source-target constraints (see Section 3.1), but at that moment we did not worry too much about the target constraints, i.e., those that define the well-formed rules of any `DocBook` model—basically because they were ensured by the forward transformation when creating the target model. Similar to the ones that were specified for the `BibTeX` models, the constraints for the target model of the Tract that we shall use in this paper are the following:

```
context PersonD inv hasToBeAuthorOrEditor:
  self.chapter->size() + self.bookE->size() > 0

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

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

context Book inv withinBookUniqueTitle:
  self.chapter->forAll( c1, c2 |
    c1 <> c2 implies c1.title <> c2.title )

context Book inv hasAuthorXorIsProc:
  self.author->isEmpty() xor self.editor->isEmpty()

context Book inv normalBookSectionsWrittenByAuthors:
  self.author->notEmpty() implies
    self.chapter->forAll(c|c.author = self.author)
```

Note that the invariant analogous to the `hasAtLeastOnePaper` invariant for Proceedings is specified by the multiplicity of the association between `Book` and `Chapter`, demanding that books should have at least one chapter. Note as well that the constraint that states that the title of an article in a proceedings cannot coincide with the title of the proceedings does not necessarily hold for books.

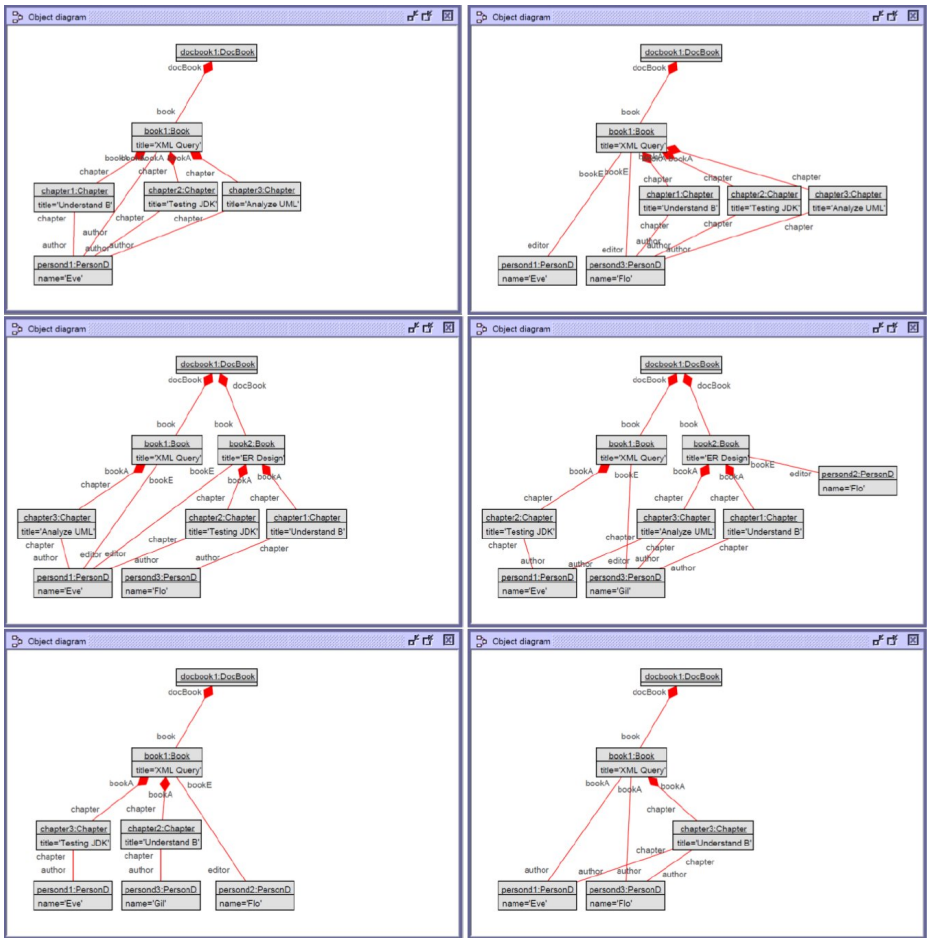


Fig. 12 The six solutions found by the model validator for the target classifying terms.

5.4 Testing BX with Classifying Terms

In Section 4.2 we discussed how to test a directional transformation using CTs, i.e., we just focused on the forward transformation. Let us discuss here how classifying terms can be used to test and understand the behavior of both directions of the BX. The main process is as follows.

1. First, we need to identify the (kinds of) models of interest in the source and target model spaces by means of classifying terms that define the corresponding equivalence classes. In our case study, they were `noSelfEditedPaper`, `noManusManumLavat` and `yearE_EQ_yearP` for the `BibTex` domain; and `noSelfEditedPaper`, `onlyNormalBooks` and `noRepeatedAuthors` for the `DocBook` domain (see Section 4.2).
2. Then, the USE model validator is used to generate the Test Suites (i.e., source and target model samples) for the Tract using these CTs. As mentioned earlier, these classifying terms define 8 equivalence classes for the source model space and 6 equivalence classes

for the target, that constitute a complete and disjoint partition of these two model spaces. The representative elements of these classes were shown in Fig. 8 and 12.
3. For the forward transformation, we execute it (employing an available BX transformation engine like MediniQVT or JTL [11]) for each model m in the source Tract suite, to get the corresponding $R(m)$ in the target, and then check whether the pair $< m, R(m) >$ conforms to the Transformation Model defined by the Tract, i.e., that it conforms to all Tract source, target and source-target constraints. Note that such a pair should be consistent according to the BX, since it has been created by the transformation with this goal in mind.
4. We also need to check whether the classifying term of every $R(m)$ in the target is the expected one.
5. We repeat the analysis now for the transformed models. Namely, we start with the set of $R(m)$ models, and apply the model transformation to check whether we get the corresponding source models m. As discussed previously (and depicted in Fig. 11) we could get that $R^{-1}(R(m)) = m$. But in case $R^{-1}(R(m)) \neq m$, we then need to check whether at least the values of the classifying terms for both m and $R^{-1}(R(m))$ coincide.
6. Working similarly for the target, we then execute the backward transformation using each model n in the target sample, to get the corresponding $R^{-1}(n)$ in the source, and check whether the pair $< R^{-1}(n), n >$ conforms to the Transformation Model defined by the Tract, i.e., that it conforms to all Tract source, target and source-target constraints.
7. We then need to check whether the values of classifying terms for $R^{-1}(n)$ in the source are the expected ones.
8. Finally, we repeat the analysis for the transformed models by the backward transformation. Namely, we start with the set of $R^{-1}(n)$ models in the source, and apply the forward model transformation to check whether we get the corresponding target models n we started with. As discussed previously (and depicted in Fig. 11) we could get that $R(R^{-1}(n)) = n$. But in case $R(R^{-1}(n)) \neq n$, then we need to check whether at least the classifying terms for both n and $R(R^{-1}(n))$ coincide.

Tables 1 and 2 summarize our findings for the `BibTex2DocBook` example, using a QVT-R transformation that we wrote in Medini-QVT and another in JTL for relating both metamodels.

Table 1 starts with the 8 models that represent the 8 equivalence classes $(m_1, ..., m_8)$ and then shows the tuple that represents their equivalence classes. The tuples with three values in the columns follow the same convention used in Fig. 10: 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 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). Column `TGT` shows the equivalence class of the transformed model $R(m)$. The following three columns (under the common heading `SRC:`$R^{-1}(R(m))$) show the equivalence class of the reconciled model in the source; whether the model $R^{-1}(R(m))$ coincides with m or not, and whether the equivalence class of $R^{-1}(R(m))$ coincides with that of m or not. The final three columns (under the common heading `TGT:`$R(R^{-1}(R(m)))$) show the behavior of the BX after the reconciliation. As we can see, once the models are reconciled, the transformation keeps them in sync — something which is expected because of the correctness and hippocraticness of the transformation.

Table 1 Result of the forward testing process.

SRC Solutions	**SRC** $[m]$	**TGT** $[R(m)]$	**BX Model Transf. Language**	**SRC:** $R^{-1}(R(m))$ $[R^{-1}(R(m))]$	$= m$?	$= [m]$?	**TGT:** $R(R^{-1}(R(m)))$ $[R(R^{-1}(R(m)))]$	$= R(m)$?	$= [R(m)]$?
m_4	0, 0, 0	0, 0, 1	MediniQVT w/o Tr	0, 0, 1	N	N	0, 0, 0	Y	Y
			MediniQVT w/ Tr	— " —	Y	Y	— " —	Y	Y
			JTL	— " —	Y	Y	— " —	Y	Y
m_8	0, 0, 1	0, 0, 1	MediniQVT w/o Tr	0, 0, 1	N	Y	0, 0, 1	Y	Y
			MediniQVT w/ Tr	— " —	Y	Y	— " —	Y	Y
			JTL	— " —	Y	Y	— " —	Y	Y
m_1	0, 1, 0	0, 0, 1	MediniQVT w/o Tr	0, 1, 1	N	N	0, 0, 0	Y	Y
			MediniQVT w/ Tr	— " —	Y	Y	— " —	Y	Y
			JTL	— " —	Y	Y	— " —	Y	Y
m_5	0, 1, 1	0, 0, 0	MediniQVT w/o Tr	0, 1, 1	N	Y	0, 0, 1	Y	Y
			MediniQVT w/ Tr	— " —	Y	Y	— " —	Y	Y
			JTL	— " —	Y	Y	— " —	Y	Y
m_6	1, 0, 0	1, 0, 1	MediniQVT w/o Tr	1, 0, 1	N	N	1, 0, 0	Y	Y
			MediniQVT w/ Tr	— " —	Y	Y	— " —	Y	Y
			JTL	— " —	Y	Y	— " —	Y	Y
m_7	1, 0, 1	1, 0, 1	MediniQVT w/o Tr	1, 0, 1	N	Y	1, 0, 1	Y	Y
			MediniQVT w/ Tr	— " —	Y	Y	— " —	Y	Y
			JTL	— " —	Y	Y	— " —	Y	Y
m_2	1, 1, 0	1, 0, 0	MediniQVT w/o Tr	1, 1, 1	N	N	1, 0, 1	Y	Y
			MediniQVT w/ Tr	— " —	Y	Y	— " —	Y	Y
			JTL	— " —	Y	Y	— " —	Y	Y
m_3	1, 1, 1	1, 0, 1	MediniQVT w/o Tr	1, 1, 1	N	Y	1, 0, 0	Y	Y
			MediniQVT w/ Tr	— " —	Y	Y	— " —	Y	Y
			JTL	— " —	Y	Y	— " —	Y	Y

Table 2 Result of the backward testing process.

TGT Solutions	**TGT** $[n]$	**SRC** $[R^{-1}(n)]$	**BX Model Tranf. Language**	**TGT:** $R(R^{-1}(n))$ $[R(R^{-1}(n))]$	$= n$?	$= [n]$?	**SRC:** $R^{-1}(R(R^{-1}(n)))$ $[R(R^{-1}(R(n)))]$	$= R^{-1}(n)$?	$= [R^{-1}(n)]$?
n_2	0, 0, 0	0, 1, 1	MediniQVT w/o Tr	0, 0, 0	Y	Y	0, 1, 1	Y	Y
			MediniQVT w/ Tr	— " —	Y	Y	— " —	Y	Y
			JTL	— " —	Y	Y	— " —	Y	Y
n_3	0, 0, 1	0, 1, 1	MediniQVT w/o Tr	0, 0, 1	Y	Y	0, 1, 1	Y	Y
			MediniQVT w/ Tr	— " —	Y	Y	— " —	Y	Y
			JTL	— " —	Y	Y	— " —	Y	Y
n_4	1, 0, 0	1, 1, 1	MediniQVT w/o Tr	1, 0, 0	Y	Y	1, 1, 1	Y	Y
			MediniQVT w/ Tr	— " —	Y	Y	— " —	Y	Y
			JTL	— " —	Y	Y	— " —	Y	Y
n_5	1, 0, 1	1, 1, 1	MediniQVT w/o Tr	1, 0, 1	Y	Y	1, 1, 1	Y	Y
			MediniQVT w/ Tr	— " —	Y	Y	— " —	Y	Y
			JTL	— " —	Y	Y	— " —	Y	Y
n_1	1, 1, 0	Incomplete model	MediniQVT w/o Tr	Incomplete model			—		
		Incomplete model	MediniQVT w/ Tr	Y			Incomplete model		
		Incorrect model	JTL	Y			Model does not conform to the metamodel		
n_6	1, 1, 1	Incomplete model	MediniQVT w/o Tr	Incomplete model			—		
		Incomplete model	MediniQVT w/ Tr	Y			Incomplete model		
		Incorrect model	JTL	Y			Model does not conform to the metamodel		

Medini-QVT offers the possibility of using or not the trace model (if it already exists, otherwise it is created) when executing a transformation, while JTL always uses it. For every source model m, we can see in Table 1 that there are three rows, each one corresponding to the results after executing the transformation using Medini-QVT without traces, Medini-QVT with traces and JTL.

Focusing on Medini-QVT without using the trace model, we can see that no model coincides with the original source, and only in some of them the equivalence classes are the same after applying the forward and backward transformations. This is as expected because there is missing information in the target models (the years of the conference), which is returned as `oclUndefined` by Medini-QVT. Please note that this is a Medini-QVT specific behavior, other model transformation engines may treat these cases differently.

When using the Medini-QVT trace model the results are different. Model $R^{-1}(R(m))$ is always m and $R(R^{-1}(R(m)))$ is always $R(m)$. The same happens with JTL.

Although in our case the results using JTL are the same as those obtained by Medini-QVT using the trace model, the trace models are different. Let us assume we have a source model m and we obtain $R(m)$ from scratch. Let us also assume that accidentally m is partially deleted and there are some `Proc` instances missing. Executing the backward transformation to the model given by $R(m)$ and using the trace model, Medini-QVT is able to restore the missing instances but not their attributes `yearE` and `yearP`. However, JTL is able to obtain the initial model m as it was originally. This is because the Medini-QVT trace model is only used for synchronization purposes, i.e., in order to not re-create target model elements if generated in previous transformations and in order to delete model elements if generated in previous transformations—but not by the current transformation. In turn, JTL's trace model is more complete as it can be used to detect those elements that are not involved in the mapping between the source and target models, forcing the transformation to write them in the model that is being generated.

In turn, Table 2 represents the behavior of the BX when we start from the six models that the model validator selected as representative elements of the corresponding classifying terms of the target model space, and which were depicted in Fig. 12.

The behavior of the QVT backward transformation with $n_2, ..., n_5$ is rather homogeneous, because the transformation works in a bijective way with them. However, for the two models (n_1, n_6) that represent normal books (i.e., those that do not have editors and hence do not have a corresponding model in the source) different implementations work in different ways.

Medini-QVT transforms them into a model that conforms to the source metamodel but that is incomplete: given that the source metamodel does not permit the existence of a Proceeding with no editors, every Book with editors is transformed into a Proceeding while the rest of the Books in the model are not transformed at all. This behavior corresponds to the one showed for model `Y` in Fig. 11. The problem is that no warning is raised by the transformation, which silently omits model elements during the transformation process. From that moment on, such elements are lost and the user will not be warned about this.

Therefore we emphasize the importance of checking the behavior of the transformation against its specifications. This issue is identified in our approach because some of the source-target constraints are violated, in particular those that state that the number of Proceedings (in the source model) and Books (in the target) should always coincide (see Sect. 3.1).

In turn, JTL creates a Proceeding from every Book although it does not have editors. As a consequence, the resulting model does not conform to the source metamodel—i.e., it is incorrect. Fig. 13 illustrates the models generated by JTL (on the left) and Medini-QVT (on the right) after executing $R^{-1}(n_6)$.

In summary, using classifying terms, we have been able to identify certain *classes* of models of interest in both the source and target model spaces and to identify the behavior of a given implementation of the bidirectional transformation in an easy manner. Given the complex behavior of any BX, these kinds of analyses are useful to spot unforseen behaviors of the transformation or potential problems when propagating or reconciling changes.

Another benefit of this testing approach is that it is easily automatable. Starting with the Tract specification, the classifying terms for the source and target model domains and the implementation of a BX, the two tables presented in Tables 1 and 2 can be automatically built using the Tract tools [5] following the process described above.

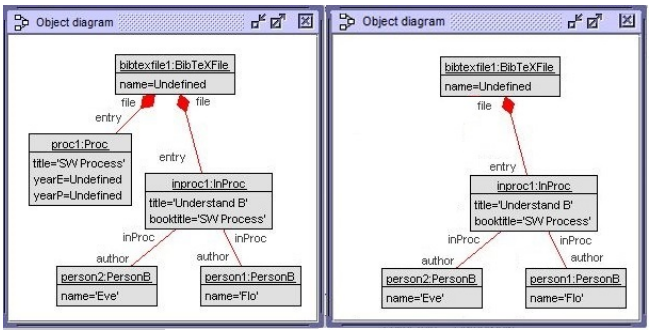


Fig. 13 Models (JTL on the left and Medini-QVT on the right) for $R^{-1}(n_6)$

6 Related Work

The work presented in this paper is a revised and enhanced version of our original paper [26], extended to improve the testing process of model transformations using classifying terms, and to cover the testing of bidirectional transformations. In order to compare our contribution to similar works, we first present related approaches which are dedicated to generate object models in a (semi-)automated manner, and then we discuss related work considering approaches for testing and verifying model transformations.

6.1 Generating Object Models

The USE model validator, used in this work, is based on the transformation of UML and OCL into relational logic [35]. 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) [23], 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 [8], relational logic and Alloy [2], term rewriting with Maude [41] or graph grammars [17]. In contrast to the tool used in this work, these approaches either do not support full OCL (e.g., higher-order associations [2] or recursive operation definitions [8] are not supported) or do not facilitate full OCL syntax checks [41]. 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 [40], on the basis of relational logic and pure Alloy [2] using a subset of OCL, and in [46] focusing on model inconsistencies by employing Kodkod. A classification of model checkers with respect to verification tasks can be found in [20].

The idea of classifying terms has similarities to the analysis of invariant independence [24]. 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.

6.2 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, [13,1]). Regarding dynamic approaches, for which the model transformation execution is needed and therefore input models, the authors in [31] and [49] present their contribution for debugging model transformations. Also, the work in [3] analyse the execution traces between the source and target models in order to find errors, and in [27] a white-box test model generation approach for testing the transformations is proposed. In this context, Tracts [48] 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 [30] that allows the specification of contracts in a visual manner, and [21] 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 [22] 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 [39] is available and presented in [10]. 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.

Equivalence partitioning [7] 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 [4] 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 [28], 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.

We recently started using Classifying Terms to check the specifications of a transformation, independently from any implementation [32]. The idea is to use the the completion capabilities of the USE model validator to simulate possible behaviours of *any* valid implementation of the transformation, and then check whether these possible behaviors are indeed acceptable.

Finally, BX testing is still a widely unexplored area of research. Most of the existing work focus on establishing the properties that a BX should exhibit to provide a *sensible* behavior. In this area, prominent works include those by Stevens [43,45], Foster [18] or Dinski [14–16]. They do not, however, aim at checking that the behavior of a BX conforms to its specifications as we focus on in this work.

Both Stevens [44] and Foster [19] have also used equivalence classes for characterizing the behaviour of bidirectional transformations. The former author defines the equivalence relations on the sets of models which are related by the transformation, i.e., the models

which are indistinguishable from the other side. The latter author defines *quotient lenses*, i.e., bidirectional transformations that are well-behaved modulo equivalence relations controlled by the programmer. That is, the equivalence classes *collapse* those model elements that should be indistinguishable from the point of view of the lenses. Both works are of a finer grain than ours: while they aim at identifying or characterizing the elements that should be treated equally by the BX, our focus is on classes that represent specific patterns (or types of elements) of particular relevance to the modeler who is interested in analysing the behavior of the transformation.

7 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, and shown how they can be effectively used in combination with the Tract specification approach for testing both directional and bidirectional 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. Particular tool support for model transformations with different options for source and target is also needed. Last but not least, classifying terms could also be used for testing model transformations specifications (i.e., *transformation models*) and not only implementations, as we have initially outlined in [32]

Acknowledgements We would like to thank Romina Eramo and Alfonso Pierantonio for their help and support with JTL. This work was partially funded by Spanish Research Project TIN2014-52034-R.

References

1. Amrani, M., Syriani, E., Wimmer, M. (eds.): Proc. of the VOLT WS., *CEUR WS. Proc.*, vol. 1325 (2014)
2. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On Challenges of Model Transformation from UML to Alloy. Software and System Modeling **9**(1), 69–86 (2010)
3. Aranega, V., Mottu, J.M., Etien, A., Dekeyser, J.L.: Traceability mechanism for error localization in model transformation. In: Proc. of ICSOFT'09 (2009)
4. Baudry, B., Dinh-Trong, T., Mottu, J., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y.: Model transformation testing challenges. In: ECMDA WS. on Integration of MDD and Model Driven Testing (2006)
5. Burgueño, L., Wimmer, M., Troya, J., Vallecillo, A.: Tractstool: Testing model transformations based on contracts. In: Joint Proc. of MODELS'13 Demos, Posters, Student Research Competition, *CEUR Workshop Proceedings*, vol. 1115, pp. 76–80. CEUR-WS.org (2013)

6. Burgueño, L., Wimmer, M., Troya, J., Vallecillo, A.: Static Fault Localization in Model Transformations. IEEE Transactions on Software Engineering **41**(5), 490–506 (2015)
7. Burnstein, I.: Practical Software Testing. Springer-Verlag (2003)
8. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models using Constraint Programming. In: Proc. of ASE'07, pp. 547–548. ACM (2007)
9. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: Proc. of the OCL and Model Driven Engineering Workshop (2004)
10. Ciancone, A., Filieri, A., Mirandola, R.: MANTra: Towards model transformation testing. In: Proc. of QUATIC'10, pp. 97–105. IEEE (2010)
11. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: JTL: a bidirectional and change propagating transformation language. In: Proc. of SLE'10, no. 6563 in LNCS, pp. 183–202. Springer (2011). `http://jtl.di.univaq.it/`
12. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: A cross-discipline perspective. In: Proc. of ICMT 2009, no. 5563 in LNCS, pp. 260–283. Springer (2009)
13. Dingel, J., de Lara, J., Lucio, L., Vangheluwe, H. (eds.): Proc. of the AMT WS., *CEUR WS. Proc.*, vol. 1277 (2014)
14. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: Proc. of MODELS'08, *LNCS*, vol. 5301, pp. 21–36. Springer (2008)
15. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations: the asymmetric case. Journal of Object Technology **10**, 6: 1–25 (2011)
16. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In: Proc. of MODELS'11, *LNCS*, vol. 6981, pp. 304–318. Springer (2011)
17. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Software and System Modeling **8**, 479–500 (2009)
18. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3) (2007)
19. Foster, N., Pilkiewicz, A., Pierce, B.C.: Quotient lenses. SIGPLAN Not. **43**(9), 383–396 (2008)
20. Gabmeyer, S., Brosch, P., Seidl, M.: A Classification of Model Checking-Based Verification Approaches for Software Models (2013). Proc. of the 1st VOLT Workshop
21. García-Domínguez, A., Kolovos, D.S., Rose, L.M., Paige, R.F., Medina-Bulo, I.: EUnit: a unit testing framework for model management tasks. In: Proc. of MODELS'11, no. 6981 in LNCS, pp. 395–409. Springer (2011)
22. Giner, P., Pelechano, V.: Test-driven development of model transformations. In: Proc. of MODELS'09, *LNCS*, vol. 5795, pp. 748–752. Springer (2009)
23. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Software and Systems Modeling **4**(4), 386–398 (2005)
24. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, independence and consequences in UML and OCL models. In: Proc. of TAP'09, *LNCS*, vol. 5668, pp. 90–104. Springer (2009)
25. Gogolla, M., Vallecillo, A.: *Tract*able model transformation testing. In: Proc. of ECMFA'11, no. 6698 in LNCS, pp. 221–236. Springer (2011)
26. Gogolla, M., Vallecillo, A., Burgueño, L., Hilken, F.: Employing Classifying Terms for Testing Model Transformations. In: Proc. of MODELS'15, pp. 312–321. IEEE (2015)
27. González, C.A., Cabot, J.: ATLTest: a white-box test generation approach for ATL transformations. In: Proc. of MODELS'12, *LNCS*, vol. 7590, pp. 449–464. Springer (2012)
28. González, C.A., Cabot, J.: Test Data Generation for Model Transformations Combining Partition and Constraint Analysis. In: Proc. of ICMT'14, *LNCS*, vol. 8568, pp. 25–41. Springer (2014)
29. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., Santos, O.: Engineering model transformations with transML. Software and Systems Modeling **12**(3), 555–577 (2013)
30. Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Automated verification of model transformations based on visual contracts. Autom. Softw. Eng. **20**(1), 5–46 (2013)
31. Hibberd, M., Lawley, M., Raymond, K.: Forensic debugging of model transformations. In: Proc. of MODELS'07, *LNCS*, vol. 4735, pp. 589–604. Springer (2007)
32. Hilken, F., Burgueño, L., Gogolla, M., Vallecillo, A.: Iterative Development of Transformation Models by Using Classifying Terms. In: Proc. of AMT'15, *CEUR Workshop Proceedings*, vol. 1500, pp. 1–6. CEUR-WS.org (2015)
33. Hu, Z., Mu, S., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. Higher-Order and Symbolic Computation **21**(1-2), 89–118 (2008)

34. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
35. Kuhlmann, M., Gogolla, M.: From UML and OCL to relational logic and back. In: Model Driven Engineering Languages and Systems, *LNCS*, vol. 7590, pp. 415–431. Springer (2012)
36. Lämmel, R.: Coupled software transformations (extended abstract). In: First International Workshop on Software Evolution Transformations (2004)
37. Meyer, B.: Applying design by contract. IEEE Computer **25**(10), 40–51 (1992)
38. Mu, S.C., Hu, Z., Takeichi, M.: An algebraic approach to bi-directional updating. In: Proc. of APLAS 2004, no. 3302 in LNCS, pp. 2–18. Springer (2004)
39. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation. Version 1.2. Object Management Group (2015)
40. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. Data Knowl. Eng. **73**, 1–22 (2012)
41. Roldán, M., Durán, F.: Dynamic Validation of OCL Constraints with mOdCL. ECEASST **44** (2011)
42. Stevens, P.: A landscape of bidirectional model transformations. In: Proc. of GTTSE'07, no. 5235 in LNCS, pp. 408–424. Springer (2007)
43. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Software and System Modeling **9**(1), 7–20 (2010)
44. Stevens, P.: Observations relating to the equivalences induced on model sets by bidirectional transformations. ECEASST **49** (2012)
45. Stevens, P.: A simple game-theoretic approach to checkonly QVT relations. Software and System Modeling **12**(1), 175–199 (2013)
46. Straeten, R.V.D., Puissant, J.P., Mens, T.: Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In: ECMFA, *LNCS*, vol. 6698, pp. 69–84. Springer (2011)
47. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Proc. of TACAS'07, pp. LNCS 4424, 632–647 (2007)
48. Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal specification and testing of model transformations. In: Formal Methods for Model-Driven Engineering (SFM). Springer (2012)
49. Wimmer, M., Kappel, G., Schönböck, J., Kusel, A., Retschitzegger, W., Schwinger, W.: A Petri Net based debugging environment for QVT Relations. In: Proc. of ASE'09 (2009)