Testing Transformation Models Using Classifying Terms

Loli Burgueño[1], Frank Hilken[2], Antonio Vallecillo[1], and Martin Gogolla[2]

[1] Universidad de Málaga, Spain {loli,av}@lcc.uma.es
[2] University of Bremen, Germany {fhilken,gogolla}@informatik.uni-bremen.de

Abstract. Testing the correctness of the specification of a model transformation can be as hard as testing the model transformation itself. Besides, this test has to wait until at least one implementation is available. In this paper we explore the use of tracts and classifying terms to test the correctness of a model transformation specification (a transformation model) on its own, independently from any implementation. We have validated the approach using two experiments and report on the problems and achievements found concerning conceptual and tooling aspects.

1 Introduction

In general, the specification of a model transformation (as of any program) can become as complex as the program itself and thus can contain errors. Normally specifications and implementations work hand-by-hand, since they both can be considered as two complementary descriptions of the intended behavior of the system, at different levels of abstraction. Checking that the implementation conforms to the specification is one possible test for correctness of both artefacts, since they are normally developed by different people, at different times, and using different languages and approaches (e.g., declarative vs imperative). However, this process needs to wait until both the specification and the implementation are available.

Unlike other approaches, such as [2], where specifications and implementations of model transformations (MT) are tested for correctness against each other, in this paper we explore the possibility of testing the correctness of the specification of a MT on its own, independently from any possible implementation. For this we use a particular contract-based approach for MT specification [9], whereby a transformation specification is modularly given by a set of tracts, each one focusing on a particular use case of the transformation. Every tract is defined in terms of particular input and output models (those relevant to the use case) and how they should be related by the transformation. Developers are then expected to identify the transformation scenarios of interest (each one defined by one tract) and check whether the transformation behaves as expected in these scenarios.

To check the correctness of a MT specification (w.r.t. the intent of the specifier) the idea is to automatically simulate the behavior of any MT that conforms

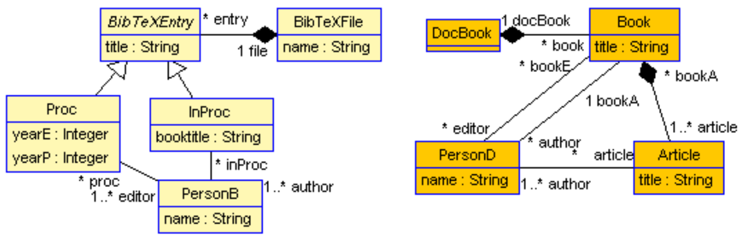

Fig. 1. BiBTeX2DocBook source and target metamodels.

to that specification, and make sure that behavior is possible (satisfiability), univocally defined (functionality) and behaves as expected (given kinds of source models are transformed into the expected kinds of target models). For this we combine the use of tracts (which provide modular pieces of specification of a model transformation) and classifying terms (CT) [14] (which permit generating relevant sample models for a tract) with the completion capabilities of the USE model validator [8]. In a nutshell, the idea is to test that a tract specification of a MT is correct by generating a set of source models, use the model validator to automatically generate target models that fulfill the source-target and target constraints of the tract, and study the properties of these automatically generated models. For example, the fact that no target models can be generated means that the Tract specification is not satisfiable. If more than one target model can be obtained for the same source model, it means that the transformation is not uniquely defined. Furthermore, we can use CTs in the target domain to check that the source models are transformed according to the specifiers expectations—or that certain kinds of target models can never be generated by the transformation.

The structure of this paper is as follows. Section 2 presents the background work on which our proposal is based: tracts, CTs and the USE model validator completion capabilities. Section 3 introduces our proposal, showing how transformation models can be tested. Then, Sections 4 and 5 describe two validation exercises we have conducted to estimate its benefits and limitations. Section 6 compares our work to similar related proposals. Finally, Section 7 concludes the paper and outlines some future work.

2 Preliminaries

2.1 A Running Example

In order to illustrate our proposal, let us consider a simple model transformation, `BiBTex2DocBook`, that converts the information about proceedings of conferences (in `BibTeX` format) into the corresponding information encoded in `DocBook` format. This example was originally presented and discussed in [14]. The source and target metamodels that we use for the transformation are shown in Fig. 1. Associated to any metamodel there are always some constraints that define the well-formed rules for their models. The constraints for the source and target metamodels of the example are shown below.

```
-- SOURCE MM (BIBTEX) CONSTRAINTS
context Person inv uniqueName: Person.allInstances->isUnique(name)
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)
-- TARGET MM (DOCBOOK) CONSTRAINTS
context PersonD inv hasToBeAuthorOrEditor:
    self.article->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.article->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.article->forAll(c|c.author = self.author)
```

2.2 Tracts

Tracts were introduced in [9] as a specification and black-box testing mechanism for model transformations. 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 are specified in OCL and define the constraints on the *source* and *target* metamodels that determine the scenario of the tract, and the *source-target* constraints that provide the *specification* of the model transformation, which in our case constitutes the *Transformation Model* (TM).

Tracts constraints for the example. For the source, we will focus on BibTeX files in which all Proceedings have at least one paper, and in which all persons are either editors or authors of an entry.

```
context Person inv isAuthorOrEditor:
    inProc->size() + proc->size() > 0
context Proc inv hasAtLeastOnePaper:
    InProc.allInstances->exists(pap | pap.booktitle=title)
```

Similarly, for the target constraints we concentrate on books with at least one article, and also require that all persons participate in an entry.

```
context Book inv AtLeastOneArticle:
    self.article->size() > 0
```

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

Finally, the Tract source-target constraints constitute the core of the TM, and specify the relationship between the two metamodels (note that these constraints are normally direction-neutral). They make use of a class `Tract` that is added to the metamodel to store all attributes and relations of the tract itself.

```
context Tract inv Proc2Book:
  self.file.entry->selectByType(Proc)->size() =
      self.docBook.book->size() and
  self.file.entry->selectByType(Proc)->forAll(proc|
    self.docBook.book->one(book|
            proc.title = book.title and
      proc.editor->size() = book.editor->size() and
      proc.editor->forAll(editorP | book.editor->
                     one(editorB|editorP.name = editorB.name))))
context Tract inv InProc2Article:
  self.file.entry->selectByType(InProc)->size() =
        self.docBook.book.article->size() and
    self.file.entry->selectByType(InProc)->forAll(inproc|
        self.docBook.book->one(b | b.title = inproc.booktitle and
              b.article->one(art|art.title = inproc.title and
      inproc.author->size() = art.author->size() and
      inproc.author->forAll(authP|art.author->
        one(authA|authP.name = authA.name)))))
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()))
```

2.3 Classifying terms

Usual approaches to generate object models from a metamodel explore the state space looking for different solutions. The problem is that many of these solutions are in fact very similar, only incorporating small changes in the values of attributes and hence “equivalent” from a conceptual or structural point of view.

Classifying terms (CT) [14] constitute a technique for developing test cases for UML and OCL models. CTs are arbitrary OCL expressions on a class model that calculate a characteristic value for each object model. The expressions can either be boolean, allowing to define up to two equivalence classes, or numerical, where each resulting number defines one equivalence class. Each equivalence class is then defined by the set of object models with identical characteristic values and with one canonical representative object model. Hence, the resulting set of object models is composed from one object model per class, and therefore they represent significantly different test cases, and partition of the full input space.

For example, the following three CTs can be defined for the source metamodel of the `BiBTeX2DocBook` transformation:

```
[ yearE_EQ_yearP ]
    Proc.allInstances->forAll(yearE=yearP)
[ noManusManumLavat ]
    not PersonB.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))
```

Each of the three CTs may be true or false. Together, there are thus eight (=2^3) equivalence classes. Each class contains those models that satisfy or not one of these properties: TTT, TTF, TFT, TFF, FTT, etc. (T if that CT evaluates to `true` and F if it evaluates to `false`). The model validator will simply return one representative of each class.

Note that we have defined some CTs for the source metamodel because we are initially interested in generating source models for the transformation. But we could have equally defined some CTs for the target metamodel in case we also want to partition the target model space. This is very useful for two main purposes. First, if we want to check that the transformation maps a certain source equivalence class of models into a given target class. And second, if we are interested in exploring some properties of the transformation or even consider the transformation in a bidirectional manner. For more details about CTs and their usages, see [14].

Also note that CTs do not always pretend to generate models that are representative of the complete metamodel, they might be used to generate models that contain interesting features w.r.t. concrete scenarios of the transformation model.

2.4 The USE model validator

Object models are automatically generated from a set of CTs by the USE model validator, which scrolls through all valid object models and selects one representative for each equivalence class. For this, as described in [14], each CT is assigned an integer value, and the values of the CTs are stored for each solution. Using the CTs and these values, constraints are created and given to a Kodkod solver along with the class model during the validation process. The solver prunes all object models that belong to equivalence classes for which there is already a representative element.

The validator has to be given a so-called configuration that determines how the classes, associations, datatypes and attributes are populated. In particular, for every class a mandatory upper bound for the number of objects must be stated. Both the USE tool and the model validator plugin are available for download from `http://sourceforge.net/projects/useocl/`.

3 A Frame for Testing Transformation Models

One central idea in this contribution is to combine CTs with a technique that *completes* a partially specified and generated object diagram. The completion technique can be applied to the source of a transformation model in order to obtain a target. Similarly, the completion technique can be applied to the target to obtain a source. Fig. 2 show an activity diagram with the process that needs to be followed to test the specifications in any of the directions.

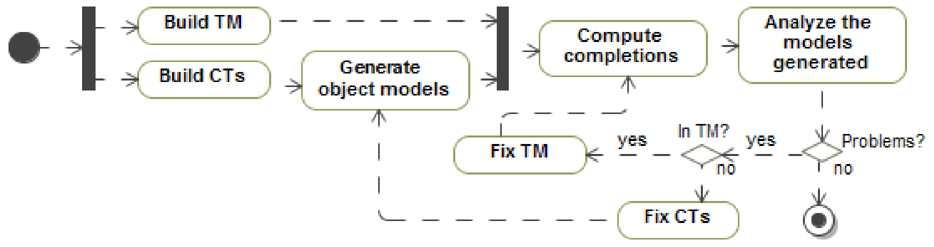

Fig. 2. Process to test the specifications.

We start building a transformation model which is constituted by a source and a target metamodel as well as the actual transformation model in the form of source-target constraints.

First, let us consider the direction from the source to the target. We build a collection of source test object models determined by some source CTs. Then we compute target completions for each source object model som, which is mapped to a collection $\{tom_1, ..., tom_n\}$ of target object models. If $n = 0$ holds, this means that the source model som cannot be completed. It reveals that either the transformation model or the source CTs and with that the test object models are inappropriately chosen.

Optionally one can now also consider the direction from the target to the source. If target CTs are available, then we can build target test object models. As before, we can compute source object models completions for a target object model tom, which is then mapped to a collection $\{som_1, ..., som_m\}$ of source object models. If we have $m = 0$, then this would mean that there is no source model for the target model tom.

The expected result is then given as follows and can be used for test purposes in various ways.

– We obtain a collection of (source,target) object model pairs that show to the developer the behavior of the transformation model in terms of concrete test cases and thus makes the transformation alive: $\{(som_1, tom_1), ..., (som_l, tom_l)\}$.
– Depending on the transformation model and the chosen CTs one will get witnesses for (a) non-functional behavior or (b) non-injective behavior of the model transformation.
  (a) Non-functional behavior of the transformation model: This occurs if there is one source object model som connected to two different target object models tom_1 and tom_2 such that (som, tom_1) and (som, tom_2) are in the (source,target) object model set with $tom_1 \neq tom_2$.
  (b) Non-injective behavior of the transformation model: This occurs if there is one target object model tom connected to two different source object models som_1 and som_2 such that (som_1, tom) and (som_2, tom) are in the (source,target) object model set with $som_1 \neq som_2$.
– Furthermore, a potentially existing connection between source and target CTs may be analysed. The developer may claim the source and the tar-

get CTs as being 'in correspondence': the collection of source object models generated from the source CTs must then be transformed into the collection of target object models generated from the target CTs. The developer can be supported in checking the 'in correspondence' claim. For each completion of a source object model, there must exist a target object model that shows the same behavior w.r.t. to the target CTs: A source object model *som* generated from the source CTs will be completed to a target object model $completed(som)$; and there must exist a target object model *tom* in the collection of target object models generated from the target CTs (CTS_{target}) such that the following is true (where the operation *eval* computes the value of the expression): $eval(CTS_{target}, completed(som)) = eval(CTS_{target}, tom)$.
If there is no such target model, the 'in correspondence' supposition is wrong and with this the CTs are inappropriately chosen, if an 'in correspondence' relationship was desired.

– In general, one can also try to consider the direction from target to source with such a correspondence checking technique.

Developing a TM for the BiBTeX2DocBook example. In order to build the MT specification (i.e., tract source-target constraints) we gave the metamodels to one experienced developer with knowledge in OCL and the description in natural language of the transformation model. Basically, each BibTeXFile should have a direct correspondence with a DocBook; each Proc with a Book; each InProc with an Article, and each PersonB with a PersonD. Moreover, all the relationships between the source objects are kept in the target model but there exists a new relationship between a book and an article when the corresponding InProc has as booktitle the corresponding Proc.

With this, the developer provided the following four constraints (one for every pair of objects) as the transformation specification.

```
context Tract inv BibTeX2DocBook:
   BibTeXFile.allInstances->forAll(file|DocBook.allInstances->exists(dB|
      file.entry->selectByType(Proc)->forAll(proc|dB.book->
        one(b|proc.title = b.title))))
context Tract inv Proc2Book:
   Proc.allInstances->forAll(proc|Book.allInstances->exists(book|
      proc.title = book.title and
      proc.editor->forAll(editorP|book.editor->exists(editorB |
         editorP.name = editorB.name and
         book.article->forAll(art|InProc.allInstances->
           one(inP | inP.booktitle = art.title))))))
context Tract inv InProc2Article:
   InProc.allInstances->forAll(inP|Article.allInstances->exists(art|
     inP.title = art.title and
     art.bookA.title = inP.booktitle and
     inP.author->forAll(authP|
       art.author->exists(authA | authP.name = authA.name))))
context Tract inv PersonB2PersonD:
   PersonB.allInstances->size() = PersonD.allInstances->size() and
   PersonB.allInstances->forAll(p|PersonD.allInstances->exists(pd|
     p.name=pd.name))
```

Given the transformation model and the object diagrams obtained using the CTs previously defined, the model validator could not complete any of the object

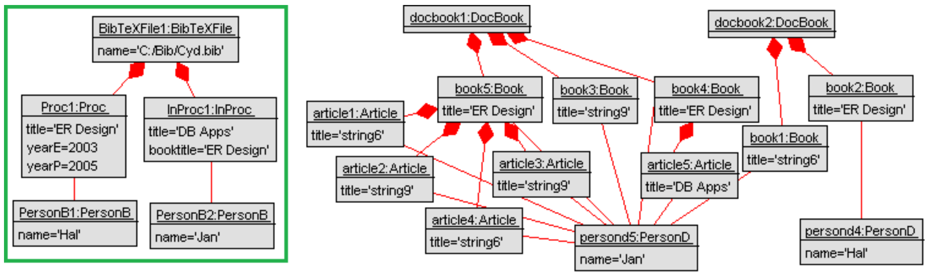

Fig. 3. Completion for the BibTeX model.

diagrams. There is no possible model that fulfils all the conditions. The reason is a fault in the constraint `Proc2Book` (line 4). At the end of it, Article titles are compared to InProc booktitles, instead of InProc titles. The fix to be applied is `inP.title = art.title`.

After fixing this error, the model validator was able to complete the source object diagrams respecting the transformation model. Figure 3 shows, inside the square, one of the source object diagrams and, outside the square, the completion generated by the model validator.

Several problems can be easily detected on this object model. First of all, there is no one-to-one correspondence between the source and target objects: the target contains more objects than it should. This problem can be solved by adding object equality to each one of the constraints to limit the number of objects generated when completing the object diagrams. For instance, for the `BibTeX2DocBook` invariant the code to add is `BibTeXFile.allInstances->size() = DocBook.allInstances->size()`.

Another observable problem is that the names were supposed to be unique (i.e., a DocBook should not have two Books with the same name and a Book should not have two Articles with the same name). In the constraints, the `one` operation, which is intended for the uniqueness, is placed inside the body of an `exists` operation. As there is one DocBook that respects the uniqueness (`docBook2`), i.e. fulfills the constraint, the overall system state is valid. In order to generate correct completions the `exists` expressions need to be replaced by `one` expressions (lines 2, 6, 13 and 20).

In summary, we have been able to follow an iterative process for the development of *transformation models*, that can be checked for correctness before any implementation is available, and independently from any of them. A transformation model was considered correct if the sample models generated from the constraints and CTs could be completed and the resulting models made sense.

4 Validation Exercise 1: Families2Persons

In order to validate our proposal with further examples and to make some estimations on the effort required by our specification development approach, we selected two representative case studies.

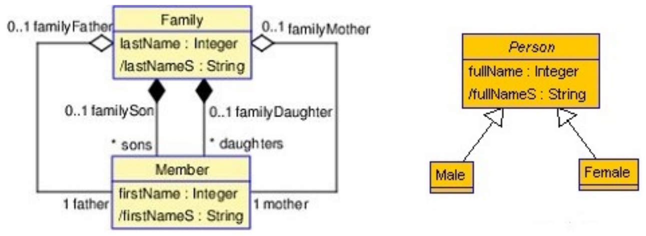

Fig. 4. Metamodels for the `Families2Persons` transformation (once corrected).

For each one we first defined a set of CTs, and created a set of test object models using them. Then, each author of this paper worked independently on developing the transformation models for both examples, testing them with the sample models jointly decided. This section and the next one describe the examples, the problems and issues found in each one, and some indications on the effort required to develop and test the corresponding transformation models.

The Families2Persons model transformation. The first case study is the well-known example of the `Families2Persons` model transformation. The source and target metamodels are depicted in Fig. 4. The validation exercise consisted in developing a TM that specifies the required model transformation. For generating the sample object models we defined one tract and two CTs. The tract focused on families with less than 4 children:

```
Family.allInstances->forAll(f|f.daughters->size + f.sons->size <= 3)
```

The constraint of the first CT identifies families with one son and one daughter. The second CT identifies families with at least three generations.

```
[oneDaughterOneSon]
   Family.allInstances->forAll(f|f.daughters->size=1 and f.sons->size=1)
[AtLeastThreeGenerations]
   Family.allInstances()->exists( fstGen |
     let sndGen = (fstGen.sons.familyFather->
                  union(fstGen.daughters.familyMother)) in
     sndGen->exists(f|f.sons->notEmpty() or f.daughters->notEmpty()))
```

The combination of these two CTs determines four equivalence classes in the input model space. Using the tract and these classifying terms, we used the model validator to build the four object models (one per equivalence class) that were used to test the transformation model.

Issues found. The first problems that we all hit when developing the tract source-target constaints were due to the fact that the original metamodels were incomplete, they even contained errors (note that Fig. 4 shows the metamodel once the problems found were fixed). These issues arose when we tried to generate object models with the model validator. In particular, in the original `Families` metamodel all relationships were compositions. This bans members from belonging to two families, and therefore the existence of families with three generations. We had to change the father and mother relationships to aggregations in order to proceed.

Furthermore, as soon as we started to generate source object models we also discovered that these metamodels permitted some pathological scenarios. For example, the original source metamodel allowed a member to play the role of father and son in the same family—similarly for mother and daughters. Analogously, members' gender should be maintained: if a person is a son in a family, he cannot be the mother in another. Names cannot be empty, either.

Finally, we also had to cope with some current limitations of the model validator: for example, it cannot deal with the `concat()` operator on strings. To overcome this limitation we used integers to represent names, instead of strings. First and last names are 1-digit integers (in decimal base) and full names are 2-digit integers that can be obtained as `name*10 + lastName`. The original string attributes became derived features, for easy human interpretation of the results.

Fig. 4 shows the final metamodel. The following invariants need to be added to the metamodels, to incorporate the corresponding well-formed constraints that permitted creating valid models.

```
-- FAMILIES METAMODEL
context Family
  inv nameDomain: 0 <= lastName and lastName <= 9
  inv fatherParadoxon: self.sons->excludes(father)
  inv motherParadoxon: self.daughters->excludes(mother)
  inv cycleFreeness: self.sons.familyFather->
     union(self.daughters.familyMother)->
       closure(f | f.sons.familyFather->
         union(f.daughters.familyMother))->excludes(self)
context Member
  inv nameDomain: 1 <= firstName and firstName <= 9
  inv noOrphanMember: self.familyMother->size() +
      self.familyFather->size() + self.familyDaughter->size()
      + self.familySon->size() > 0
  inv sonsBecomeFathers: self.familySon <> null implies
      self.familyMother = null
  inv daughtersBecomeMothers: self.familyDaughter <> null implies
      self.familyFather = null
  inv fatherMotherDistinct: self.familyFather = null or
      self.familyMother = null
-- PERSON METAMODEL
context Person
  inv nameDomain: 10 <= fullName and fullName <= 99
  inv nameNotEmpty: fullName <> null
```

When generating the sample test object models we also discovered that these metamodels permit other weird situations. For example, all members of a family can have the same first name. Similarly, the last name of family members in multiple families needs to be preserved and this is not controlled by the metamodel. In particular, the last name of a father or a son should be the same in all of his families.

Apart from these issues concerning the source and target metamodels, when trying to develop and test a transformation model using our approach, we also unveiled some issues related with the relationship that the transformation is trying to establish between the metamodels. For example, consider the following constraint that checks that members are transformed to persons with the proper gender (an equivalent one should be required for male members):

```
(self.familyDaughter <> null or self.familyMother <> null) implies
  Female.allInstances->exists(female |
```

Table 1. Statistics for the two case studies.

	Families2Persons				Java2Graph			
	AV	FH	LB	MG	AV	FH	LB	MG
# iterations	5	2	4	2	4	2	4	2
# iterations where new problems were detected	4	1	3	1	3	1	3	1
# iterations in which a fix led to a new problem	0	0	0	0	0	0	0	0
% of time developing the OCL code (vs. % of time finding what the problem was)	40	76	20	10	40	82	20	10
% of occasions the model validator returned UNSAT (instead of generating the models)	40	0	10	0	50	0	20	0
final # of constraints developed	5	1	4	3	4	4	5	2

```
    if self.familyMother <> null then
      female.fullName=(self.firstName*10)+self.familyMother.lastName
    else
      female.fullName=(self.firstName*10)+self.familyDaughter.lastName
    endif)
```

This approach does not work: should there be two members (left-hand side) with the same name and surname, the constraint (either with `exists` or with `one`) is fulfilled as long as there is one person (right-hand side) with the name and surname. And if we try to add another constraint that states that the number of members and persons should be the same, the model validator may create a person with a random `fullName` in order to enforce it. Therefore, apart from the constraint that checks the size, there is the need to check the exact correspondence of the names of members and persons. There are several approaches to solve this issue, and each author used one—e.g., navigating the set of all possible names and checking that the number of times they appear in the source and target models is the same; or selecting the members by gender and making sure their names appear the same number of times in both sides.

Performance analysis for the Families2Persons case study. Columns 2-5 of Table 1 show the general statistics resulting from the experiments conducted by the four paper authors (AV, FH, LB, MG) for this case study. Each author needed a number of iterations until his/her transformation model was developed and tested correct. As mentioned above, a transformation model was considered correct if the sample source models generated from the selected tract and CTs could be completed and the resulting target models were unique and made sense.

It is important to note that the tests were not really independent, particularly at the beginning of the experiment when the source and target metamodels were found inconsistent and incomplete. In order to focus on the transformation model itself, as soon as someone detected a problem in the metamodel, it was corrected and the rest of the participants were notified to avoid it.

The level of expertise of FH and MG with OCL and with the model validator was higher, and this fact had some impact when developing the TMs, as Table 1 reflects. They needed less iterations to solve the problems and they never wrote constraints that led to UNSAT states.

Although not specified in the table, note as well that in the first iterations, most of the time was spent developing the OCL code of the TM, while in the latter ones the problems were trickier and thus more time was spent identifying the problems and solving them. The total time spent running this experiment ranged between six and eight hours.

5 Validation Exercise 2: Java2Graph

The second validation experiment considered a model transformation that, starting from a model of a large Java program, transformed it into a graph (composed of nodes and edges) that has the information needed to be visualized. Graph nodes have name, shape, colour and size. Edges have a name and a label and connect nodes. Every edge is connected to one source and one target node, but you may have many different edges between two nodes.

In the transformation, every Java class is represented by a node, and every attribute whose type is another class as an edge. The label of the edge is the name of the attribute. The size of a node is the number of its outgoing edges. The shape depends on the kind of Java class: triangular if the class is abstract, square if it is final, and round (a circle) if it is a regular class. The colour depends on the number of attributes and methods: red if the class has more attributes and methods than 70% of the rest of the classes, green if it is between percentiles 30 and 70, and yellow if it has less than 30% of the rest of the classes.

The source and target metamodels for the transformation are shown in Fig. 5. Additionally, some constraints define their well-formed rules:

```
-- JAVA METAMODEL CONSTRAINTS
context NamedElement inv uniqueNames:
  not self.name.oclIsUndefined() and (self.name <> '')
  and NamedElement.allInstances->select(ne |
         self<>ne and ne.name = self.name)->size()=0
context Class inv kindNotNull:
  not self.kind.oclIsUndefined()
context Attribute inv typeNotNull:
  not self.type.oclIsUndefined()
context DataType inv noDanglingDatatypes:
    Attribute.allInstances()->exists(a | a.type = self)
-- GRAPH METAMODEL CONSTRAINTS
context Node inv validNode:
    (self.name <> null) and (self.name <> '') and (self.size>=0)
context Edge inv noNullLabel:
    (self.label <> null) and (self.label <> '')
```

To generate the sample object models used to test the transformation model we defined three CTs. They define eight equivalence classes and thus eight sample object models with different characteristics.

```
[ThreeKindsOfClasses]
  Class.allInstances->exists(c1, c2, c3 | c1.kind=#abstractClass
    and c2.kind=#finalClass and c3.kind=#regularClass)
[GodObjectExists]
  Class.allInstances()->exists( c | c.atts.type->asSet()
    ->selectByKind(Class)
      ->includesAll(Class.allInstances()->excluding(c)))
[AttributeCycleLength3]
  Class.allInstances()->exists(c1,c2,c3 | Set{c1,c2,c3}->size=3 and
    c1.atts->exists(a1 | c2.atts->exists(a2 | c3.atts->exists(a3 |
      a1.type=c2 and a2.type=c3 and a3.type=c1))))
```

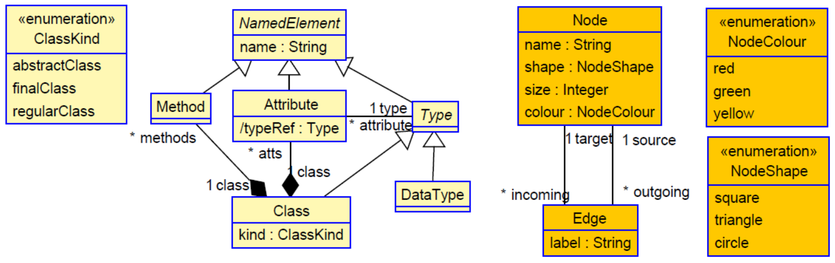

Fig. 5. Metamodels for the Java2Graph transformation.

The first CT defines models with three kinds of classes. The second one asks models to have a class that is dependent on all others. The last one requires the existence of references between classes that form cycles of length 3.

Issues found. The nature of the issues found in this example was similar to the previous one, although the number of issues was very different.

This time we did not find any major problem with the original metamodels. The only issue found with the metamodels was not due to the metamodels themselves, but to the need to relate them by a model transformation: we found the typical example of a semantic mismatch between two metamodels when we want to relate them. In the target metamodel, we had stated in Node's `validNode` invariant that the attribute `size` should be greater than 0 with the expression "`...and (self.size>0)`", and hence we had to change it to allow the attribute to be 0 by replacing it with "`...and (self.size>=0)`. Otherwise we were not able to transform Java classes with no attributes or methods, since the size of their corresponding nodes is 0.

We also hit here the problem of imprecise specifications expressed in natural language. Namely, the problem specification (imprecisely) stated that the colour of the transformed node depended on the number of attributes and methods of "the rest of the classes." One of the authors understood that in "the rest of the classes" the class self was also included while another author excluded it. This led to slightly different results in the transformed models.

Finally, we also had to overcome some limitations of the current version of the model validator, although they were easy to address. For example, it does not support real numbers and we had to use integers instead (multiplying by 100 to get 2 decimals). Furthermore, there is no support for some OCL features such as `iterate`, `sortedby` and `Sequence`. We had to express the constraints using the subset of OCL currently available.

Performance analysis for the Java2Graph case study. Columns 6-9 of Table 1 show the statistics resulting from the experiments conducted by the four authors (AV, FH, LB, MG). They are very similar to the ones obtained for the previous experiment, and the same remarks apply to them too. The time every author spent working on this experiment was between 4 and 6 hours, slightly less than before—but also because we learned from the previous experience.

Conclusions from the validation experiments. After conducting these two experiments, there are some common issues that we would like to point out. First, we were able to unveil significant problems in the specification of the source and target metamodels (even in the *a priori* very simple `Families2Persons` example). It was clear that no rigorous tests had been conducted on them, probably due to the traditional lack of tools in MDE to instantiate metamodels.

Second, we were also able to reveal semantic problems, which happen when trying to relate two (perfectly correct) independent metamodels by means of a model transformation: some of the individual metamodel integrity constraints hinder the creation of particular relations between their elements.

Third, we also suffered from the lack of precision in textual specifications, which permit multiple interpretations and hence distinct implementations.

Tool support can also represent some impediments. We hit some problems due to restricted support of model features by model validator: Java heap space when generating models with the constraint solver; lack of support for some OCL types (`Sequence`, `Real`) and operators (`iterator`, `sortedBy`).

When it comes to debugging, once the model validator finds one solution it is not easy to determine whether it is correct or not. The only thing we know is that it satisfies the imposed constraints. But then we need to interpret the results and check whether they make sense. A similar situation happens when the model validator finds more than one solution. The case is even worse when it finds no solution, because we only know that the completion is unsatisfiable.

We also learned interesting aspects of the use of the model validator. For example, using specific configurations for each test case is better than a common configuration for all cases. In this sense, a better support for model transformations and completions would be nice on the model validator side.

Finally, when we put together all the solutions independently developed by the four authors, we discovered a common schema for defining the transformation model constraints, which is the one that we show below (note that `exists` could be replaced by `one` in some cases):

```
Source->forAll(s | Target->exists(t | predicate(s,t))) -- AND --
Source->size()=Target->size()
```

Basically, in many cases the TM is composed of constraints that specify, for each source model element, how it is transformed; plus a set of constraints that determine the relations between the sizes of source and target model elements.

Note that in this paper we have focused on the validation of directional model transformations, from source to target. In general, transformation models and their associated constraints are direction neutral and thus they can be interpreted (and evaluated) from left to right and vice-versa. The validation from target to source is left for future work.

6 Related Work

In this work, we see model transformations as transformation models and focus on checking their correctness. The ideas presented in this work were first outlined in a short paper [13]. This paper contains the first complete proposal and provides some initial validation exercises that permit evaluating the advantages and shortcomings of the approach.

There are some dynamic approaches for testing MT implementations by executing them given an input model or a set of them. References [12] and [18] present contributions for debugging model transformations, and the work in [7] compares the generated and expected output models. The work in [1] analyses the trace model in order to find errors. In addition to Tracts, other static approaches allow the specification of contracts in a visual manner [11].

Reference [6] proposes a dynamic testing technique defining equivalence classes for the source models in a similar manner as it is done with CTs. Their proposal lacks full automation and is less expressive as they do not consider the use of OCL. Reference [10] presents a mechanism for generating test cases by analysing the OCL expressions in the source metamodel in order to partition the input model space. 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. Sen et al. [16] considered the completion of partial models to satisfy a set of constraints.

The work in [15] proves the correctness of specifications by making use of algebras. Our approach can be seen as a first step and as an easier and cheaper way that does not require the developer to have any extra knowledge or create any other software artifact. Similarly, in [5], Alloy is used to validate models while the UML/OCL model under test is translated to and from Alloy before and after the validation, respectively. This works for Alloy experts, but might be difficult for other modelers. Our approach works on the UML/OCL and hides the transformation details between the languages.

Finally, the work in [3] is similar to ours but uses refinement between specifications and/or implementations to check whether one can safely replace another. OCL is used as a base language where specifications and MT implementations are mapped to, and then SAT solvers look for counterexamples. As part of future work we would like to evaluate if our proposal is simpler (and more effective) than writing a reference implementation in parallel and checking if the implementation is a refinement of the specification. Refinement of MT specifications has also been studied in [17] in the context of tracts.

7 Conclusions and Future Work

Even the simplest model transformations, not to mention their specifications, may contain faults [4]. In this paper we have proposed a method that uses tracts and classifying terms to test the correctness of a model transformation specification (i.e., a transformation model), independently from any possible implementation. We have validated the approach using two experiments and report on the problems and achievements found concerning conceptual and tooling aspects.

There are several lines of work that we plan to address next. First, we want to take into account the fact that TM constraints are direction-neutral, being able to analyze further properties of the MT under development. Second, we would like to validate our proposal with more transformations, in order to gain a better understanding of its advantages and limitations. Third, we plan to improve the tool support to further automate all tests, so human intervention is kept to the minimum. Finally, we need to define a systematic approach of defining classifying term and transformation model testing using the ideas outlined in this paper.

References

1. Aranega, V., Mottu, J.M., Etien, A., Dekeyser, J.L.: Traceability mechanism for error localization in model transformation. In: Proc. of ICSOFT'09. (2009)
2. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: Static fault localization in model transformations. IEEE Trans. Software Eng. **41**(5) (2015) 490–506
3. Büttner, F., Egea, M., Guerra, E., de Lara, J.: Checking model transformation refinement. In: Proc. of ICMT'13. Volume 7909 of LNCS., Springer (2013) 158–173
4. Cuadrado, J.S., Guerra, E., de Lara, J.: Static analysis of model transformations. IEEE Trans. Software Eng. **in press**(1) (2016) 1–32
5. Cunha, A., Garis, A.G., Riesco, D.: Translating between alloy specifications and UML class diagrams annotated with OCL. SoSym **14**(1) (2015) 5–25
6. Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.L.: Qualifying input test data for model transformations. Software & Systems Modeling **8**(2) (2009) 185–203
7. 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. Volume 6981 of LNCS., Springer (2011) 395–409
8. Gogolla, M., Hamann, L., Hilken, F.: On static and dynamic analysis of UML and OCL transformation models. In: Proc. of AMT'14. CEUR Workshop Proceedings (2014) 24–33 `http://ceur-ws.org/Vol-1277/3.pdf`.
9. Gogolla, M., Vallecillo, A.: *Tract*able Model Transformation Testing. In: Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA 2011). Volume 6698 of LNCS., Springer (2011) 221–235
10. González, C.A., Cabot, J.: Test Data Generation for Model Transformations Combining Partition and Constraint Analysis. In: Proc. of ICMT'14. (2014) 25–41
11. 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) (2013) 5–46
12. Hibberd, M., Lawley, M., Raymond, K.: Forensic debugging of model transformations. In: Proc. of MODELS'07. Volume 4735 of LNCS., Springer (2007) 589–604
13. 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 (2015) 1–6 `http://ceur-ws.org/Vol-1500/paper2.pdf`.
14. Hilken, F., Gogolla, M., Burgueño, L., Vallecillo, A.: Testing models and model transformations using classifying terms. SoSyM (2016) `http://link.springer.com/article/10.1007%2Fs10270-016-0568-3`.
15. Orejas, F., Wirsing, M.: On the specification and verification of model transformations. In: Semantics and Algebraic Specification, Essays Dedicated to Peter D. Mosses on his 60th Birthday. Volume 5700 of LNCS., Springer (2009) 140–161
16. Sen, S., Mottu, J.M., Tisi, M., Cabot, J.: Using models of partial knowledge to test model transformations. In: Proc. of ICMT'12. LNCS (2012) 24–39
17. Vallecillo, A., Gogolla, M.: Typing Model Transformations Using Tracts. In: Proc. of ICMT'12. Volume 7307 of LNCS., Springer (2012) 56–71
18. Wimmer, M., Kappel, G., et al.: A Petri Net based debugging environment for QVT Relations. In: Proc. of ASE'09. (2009)