*Tract*able Model Transformation Testing

Martin Gogolla[1] and Antonio Vallecillo[2]

[1] Database Systems Group, University of Bremen, Germany
[2] GISUM/Atenea Research Group, Universidad de Málaga, Spain
gogolla@informatik.uni-bremen.de, av@lcc.uma.es

Abstract. Model transformation (MT) testing is gaining interest as the size and complexity of MTs grows. In general it is very difficult and expensive (time and computational complexity-wise) to validate in full the correctness of a MT. This paper presents a MT testing approach based on the concept of Tract, which is a generalization of the concept of Model Transformation Contract. A Tract defines a set of constraints on the source and target metamodels, a set of source-target constraints, and a tract test suite, i.e., a collection of source models satisfying the source constraints. We automatically generate input test suite models, which are then transformed into output models by the transformation under test, and the results checked with the USE tool (UML-based Specification Environment) against the constraints defined for the transformation. We show the different kinds of tests that can be conducted over a MT using this automated process, and the kinds of problems it can help uncovering.

1 Introduction

Model transformations are key elements of Model-driven Engineering (MDE). They allow querying, synthesizing and transforming models into other models or into code, and can also be composed in chains for building new and more powerful model transformations.

As the size and complexity of model transformations grow, there is an increasing need to count on mechanisms and tools for testing their correctness. This is specially important in case of transformations with hundreds or thousands of rules, for which manual debugging is no longer possible—but for which we still need to check whether the produced models conform to the target metamodel, or whether some essential properties are preserved by the transformation.

Testing model transformations is not an easy task and present numerous challenges [1,2,3,4]. In the literature there are two main approaches to model transformation testing (see also Section 5). In the first place we have the works that aim at fully *validating* the behaviour of the transformation and its associated properties (confluence of the rules, termination, etc.) using formal methods and their associated toolkits (see, e.g., [5,6,7,8,9,10,11]). The potential limitations with these proposals lies in their inherent computational complexity, which makes them inappropriate for fully testing large and complex model transformations. An alternative approach (proposed in, e.g., [12,13,14,15,16]) consists of trying to *certify* that a transformation works for a selected set of test input models, without trying to validate it for the full input space. Although such a certification approach cannot fully prove correctness, it can be very useful for identifying bugs in a very cost-effective manner, and can deal with industrial-size transformations without having to abstract away any of the structural or behavioural properties of the transformations.

In this paper we will follow this latter approach, making use of some of the concepts, languages and tools that have proved to be very useful in the case of model validation [17]. In particular, we generalize *model transformation contracts* [2,18] for the specification of the properties that need to be checked for a transformation, and then apply the ASSL language [19] to generate input test models, which are then automatically transformed into output models and checked against the set of contracts defined for the transformation, using the USE tool [20].

This paper is organized as follows. After this introduction, Section 2 describes the context of our work and introduces the running example that will be used throughout the paper to illustrate our approach. Section 3 presents our proposal and describes the prototype we have developed as a proof-of-concept. Then, Section 4 discusses the kinds of tests that can be conducted and how to perform them. Finally, Section 5 compares our work with other related proposals and Section 6 draws the final conclusions and outlines some lines for future work.

2 Context

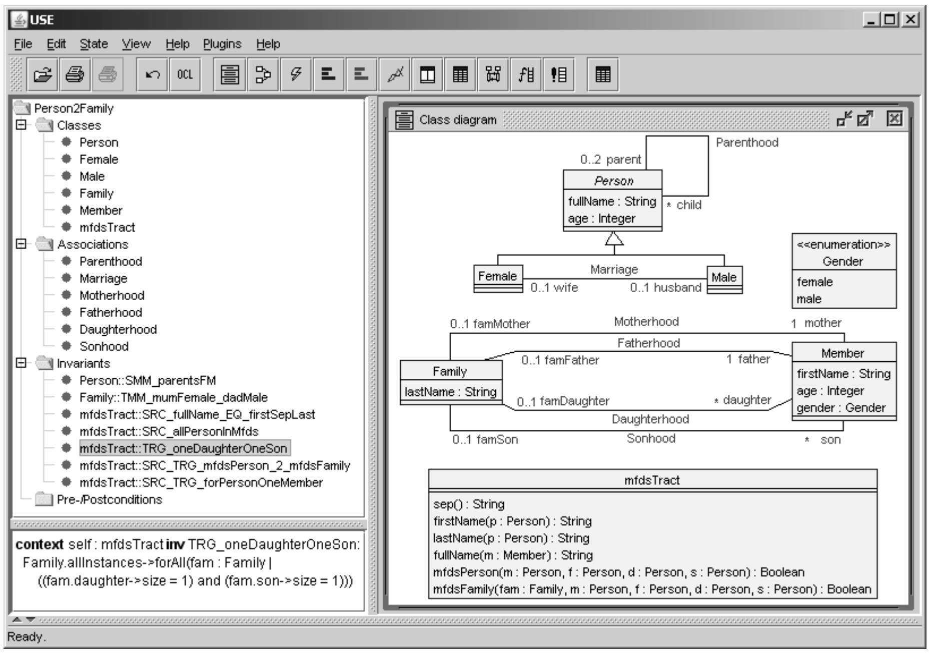
2.1 Models and Metamodels

In MDE, models are defined in the language of their metamodels. In this paper we consider that metamodels are defined by a set of classes, binary associations between them, and a set of integrity constraints.

In Figure 1 we show our running example as handled by the tool USE [20]. The aim of the example is to transform a `Person` source metamodel shown in the upper part of the class diagram into a `Family` target metamodel displayed in the middle part. The source permits representing people and their relations (marriage, parents, children) while the target focuses on families and their members.

Some integrity constraints are expressed as multiplicity constraints in the metamodels, such as the ones that state that a family always has to have one mother and one father, or that a person (either female or male) can be married to at most one person.

There are other constraints that require specialized notations because they imply more complex expressions. In this paper we will use OCL [21] as the language for stating constraints. In order to keep matters simple, we have decided to include only one source metamodel constraint (`SMM`) and one target metamodel constraint (`TMM`). On the `Person` side (source), we require that, if two parents are present, they must have different gender (`SMM_parentsFM`). On the `Family` side (target), we require an analogous condition (`TMM_mumFemale_dadMale`). Many further constraints (like acyclicity of parenthood or exclusion of marriage between parents and children or between siblings) could be stated for the two models.



```
context self : mfdsTract inv TRG_oneDaughterOneSon:
 Family.allInstances->forAll(fam : Family |
    ((fam.daughter->size = 1) and (fam.son->size = 1)))
```

Fig. 1. USE Screenshot

```
context Person inv SMM_parentsFM:
  parent->size()=2 implies
    parent->select(oclIsTypeOf(Female))->size()=1 and
    parent->select(oclIsTypeOf(Male))->size()=1
context Family inv TMM_mumFemale_dadMale:
  mother.gender = #female and father.gender = #male
```

2.2 Model Transformations

In a nutshell, a model transformation is an algorithmic specification (either declarative or operational) of the relationship between models, and more specifically the mapping from one model to another. A model transformation involves at least two models (the source and the target), which may conform to the same or to different metamodels. The transformation specification is often given by a set of model transformation rules, which describe how a model in the source language can be transformed into a model in the target language.

One of the challenges of model transformation testing is the heterogeneity of model transformation languages and techniques [4]. This problem is aggravated by the possibility of having to test model transformations which are defined as a composition of several model transformations chained together. In our proposal we use a black-box testing approach, by which a model transformation is just a program that we invoke. The main advantages of this approach are that we can deal with any transformation language and that we will be able to test the model transformation *as-is*, i.e., without having to transform it into any other language, represent it using any formalism, or abstract away any of its features.

To illustrate our proposal we will use a running example of a model transformation that, given a `Family` model, creates a `Person` model. The goal is to show how this transformation can be tested. For this place we asked some students to write such a transformation, and the resulting code is shown below. It is written in ATL [22], a hybrid model transformation language containing a mixture of declarative and imperative constructs which is widely used in industry and academia.

```
module Persons2Families;
create OUT : Families from IN : Persons;

rule Father2Family{
 from f : Persons!Male (not f.child -> isEmpty())
 to fam : Families!Family (
     lastName <-f.name.substring(f.name.lastIndexOf('␣')+2,
                                  f.name.size()) ),
    mb : Families!Member (
     firstName <- f.name.substring(1,f.name.lastIndexOf('␣')),
     age <- f.age, gender <- #male, famFather <- fam )
}
rule Mother2Family{
 from m : Persons!Female (not m.child -> isEmpty())
 to mb : Families!Member (
     firstName <- m.name.substring(1,m.name.lastIndexOf('␣')),
     age <- m.age, gender <- #female, famMother <- m.husband )
}
rule Son2Family{
 from s : Persons!Male (s.child -> isEmpty())
 to mb : Families!Member (
     firstName <- s.name.substring(1,s.name.lastIndexOf('␣')),
     age <- s.age, gender <- #male,
     famSon <-s.parent->select(e|e.oclIsTypeOf(Persons!Male)) )
}
rule Daughter2Family{
 from d : Persons!Female (d.child -> isEmpty())
 to mb : Families!Member (
     firstName <-d.name.substring(1,d.name.lastIndexOf('␣')),
     age <- d.age, gender <- #female,
     famDaughter <- d.parent->select(e|e.oclIsTypeOf(Persons!Male)) )
}
```

This transformation is defined in terms of four basic rules, each one responsible for building the corresponding target model elements depending on the four kinds of role a source person can play in a family: father, mother, son or daughter. The attributes and references of every target element are calculated using the information of the source elements. Target elements that represent families are created with the last name of the father (in rule `Father2Family`).

3 Tracts for Model Transformations

3.1 Model Transformation Contracts

In Figure 2 we have displayed the central ingredients of our approach for transformation testing: a source and target metamodel, the transformation T under test, and a transformation contract, for short *tract*, which consists of a tract test suite and a set of tract constraints. The test suite and its transformation result

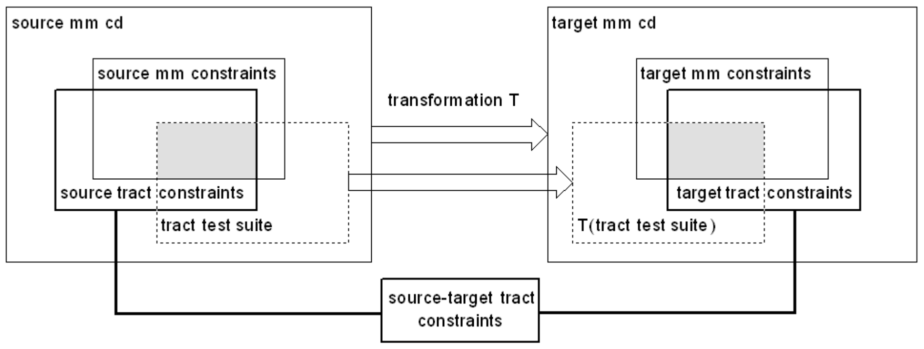


Fig. 2. Building Blocks of a Tract

are shown with dashed lines and the different tract constraints with thick lines. Five different kinds of constraints are present: the source and target class diagrams are restricted by source and target metamodels constraints, and the tract imposes source, target, and source-target tract constraints. Such constraints are expressed by means of OCL invariants. The context of these invariants is a class representing a transformation tract, a so-called tract class. An example of a tract class called `mfdsTract` is shown in Figure 1.

Assume a source model M being an element of the test suite and satisfying the metamodel source and the tract source constraints is given. Then, the tract essentially requires that the result $T(M)$ of applying transformation T satisfies the target metamodel and the target tract constraints and the pair $(M, T(M))$ satisfies the source-target tract constraints. The source-target tract constraints are crucial insofar that they can establish a correspondence between a source element and a target element in a declarative way by means of a formula. Note that this declarative correspondence between source and target has to be made explicit in other transformation approaches like the triple graph grammar (TGG) proposal [23] or the Epsilon framework [24]. In technical terms, a source tract constraint is basically an OCL expression with free variables over source elements, a target tract constraint has free variables over target elements, and a source-target tract constraint possesses free variables over source and target elements. Figure 3 gives an overview on the used concepts and their connection.

In Figure 2, the rectangles indicate possible overlap (resp. disjointness) of source and target models. Basically, the tract — consisting of the test suite and the three kinds of constraints — checks for the correctness of the transformation in the sense that correct source models from the test suite are transformed to correct target models, i.e., our approach checks that in Figure 2 the grey source section is transformed into the grey target section. In general, there will be more than one tract for a single transformation because particular source models are constructed in the test suite which then induce particular tract constraints.

Let us go back to our example in Figure 1. The lower part of the class diagram pictures the tract metamodel represented by the class `mfdsTract` where mfds is

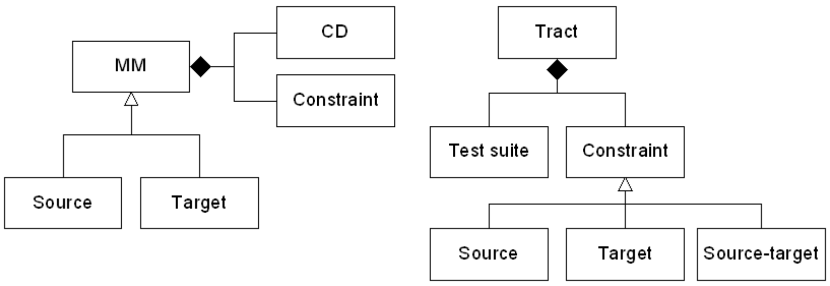


Fig. 3. Concepts in a Tract

a shortcut for mother-father-daughter-son expressing that our tract and our testing (for demonstration purposes) concentrates on conventional families with exactly one person in the respective role. The operations in class `mfdsTract` are helper operations for formulating the tract constraints which are shown as invariants on the left in the project browser. The five different kinds of constraints are reflected by different prefixes for invariant names: `SMM` for source metamodel constraints, `TMM` for target metamodel constraints, `SRC` for source tract constraints, `TRG` for target tract constraints, and `SRC_TRG` for source-target tract constraints.

Note that concepts like father or mother are not explicitly present in the `Person` metamodel (through attributes or association ends). Besides, please be warned: both metamodels and their transformation seem simple, but intricate complications live under the surface. Roughly speaking, the transformation must (*a*) split one source attribute into two target attributes in different target classes; (*b*) merge two source associations into one target class and four target associations; (*c*) map a source generalization hierarchy into a target attribute. The following listing details the five OCL invariants that constitute the `mfdsTract`.

```
inv SRC_fullName_EQ_firstSepLast:
  Person.allInstances->forAll(p|
    p.fullName=firstName(p).concat(sep()).concat(lastName(p)))
inv SRC_allPersonInMfds:
  let allFs=Female.allInstances in let allMs=Male.allInstances in
  Person.allInstances->forAll(p|
    Bag{allFs->exists(d   | allMs->exists(f,s| mfdsPerson(p,f,d,s))),
        allFs->exists(m,d| allMs->exists(s   | mfdsPerson(m,p,d,s))),
        allFs->exists(m   | allMs->exists(f,s| mfdsPerson(m,f,p,s))),
        allFs->exists(m,d| allMs->exists(f   | mfdsPerson(m,f,d,p)))} =
    Bag{true,false,false,false})
inv TRG_oneDaughterOneSon:
  Family.allInstances->forAll(fam |
    fam.daughter->size()=1 and fam.son->size()=1)
inv SRC_TRG_mfdsPerson_2_mfdsFamily:
  Female.allInstances->forAll(m,d| Male.allInstances->forAll(f,s|
    mfdsPerson(m,f,d,s) implies
    Family.allInstances->exists(fam|mfdsFamily(fam,m,f,d,s))))
inv SRC_TRG_forPersonOneMember:
  Female.allInstances->forAll(p| Member.allInstances->one(m|
    p.fullName=fullName(m) and p.age=m.age and m.gender = #female and
    (p.child->notEmpty() implies (let fam=m.famMother in
      p.child->size()=fam.daughter->union(fam.son)->size())) and
    (p.parent->notEmpty() implies m.famDaughter.isDefined()) and
    (p.husband.isDefined() implies m.famMother.isDefined()) )) and
  Male.allInstances->forAll(p| Member.allInstances->one(m|
    p.fullName=fullName(m) and p.age=m.age and m.gender = #male and
    (p.child->notEmpty() implies (let fam=m.famFather in
      p.child->size()=fam.daughter->union(fam.son)->size())) and
    (p.parent->notEmpty() implies m.famSon.isDefined()) and
    (p.wife.isDefined() implies m.famFather.isDefined()) ))
```

There are two source, one target, and two source-target tract constraints. The source constraint `SRC_fullName_EQ_firstSepLast` guarantees that one can decompose the `fullName` into a `firstName`, a separator, and a `lastName`. The source constraint `SRC_allPersonInMfds` requires that every Person appears exactly once in a `mfdsPerson` pattern. `mfdsPerson` patterns are described by the boolean operation `mfdsPerson` which characterizes an isolated mother-father-daughter-son pattern having no further links to other persons.

The constraint `SRC_allPersonInMfds` is universally quantified on `Person` objects. Each `Person` must appear either as a mother or as a father or as a daughter or as a son. This exclusive-or requirement is formulated as a comparison between bags of Boolean values. From the four possible cases, exactly one case must be true. Technically this is realized by requiring that the bag of truth values, which arises from the evaluation of the respective sub-formluas, contains exactly once the Boolean value `true` and three times the Boolean value `false`.

```
mfdsTract::mfdsPerson(m:Person,f:Person,d:Person,s:Person):Boolean=
  Set{m,f,d,s}->excluding(null)->size()=4 and
  m.oclIsTypeOf(Female) and f.oclIsTypeOf(Male) and
  m.oclAsType(Female).husband=f and
  d.oclIsTypeOf(Female) and s.oclIsTypeOf(Male) and
  m.child=Set{d,s} and f.child=Set{d,s} and
  d.parent=Set{m,f} and s.parent=Set{m,f}
mfdsTract::
  mfdsFamily(fam:Family,m:Person,f:Person,d:Person,s:Person):Boolean=
  fam.lastName=lastName(m) and fam.lastName=lastName(f) and
  fam.lastName=lastName(d) and fam.lastName=lastName(s) and
  fam.mother.firstName=firstName(m) and
  fam.father.firstName=firstName(f) and
  fam.daughter.firstName=Bag{firstName(d)} and
  fam.son.firstName=Bag{firstName(s)}
```

Both source constraints reduce the range of source models to be tested. The target tract constraint `TRG_oneDaughterOneSon` basically focusses the target on models in which the multiplicity * on the daughter and son roles are changed to the multiplicity 1. The first central source-target constraint `SRC_TRG_mfds-Person_2_mfdsFamily` demands that a `mfdsPerson` pattern must be found in transformed form as a mfds `Family` pattern in the resulting target model. The second central source-target constraint `SRC_TRG_forPersonOneMember` requires that a `Person` must be transformed into exactly one `Member` having comparable attribute values and roles as the originating Person. Both source-target tract constraints are central insofar that they establish a correspondence between a `Person` (from the source) and a `Family Member` (from the target) in a declarative way by means of a formula.

3.2 Generating Test Input Models

The generation of source models for testing purposes is done by means of the language ASSL (A Snapshot Sequence Language) [19]. ASSL was developed to generate object diagrams for a given class diagram in a flexible way. Positive and negative test cases can be built, i.e., object diagrams satisfying all constraints or violating at least one constraint. ASSL is basically an imperative programming language with features for randomly choosing attribute values or association ends. Furthermore ASSL supports backtracking for finding object diagrams with particular properties.

For the example, we concentrate on the generation of (possibly) isolated mfds patterns representing families with exactly one mother, father, daughter, and son in the respective role. The procedure `genMfdsPerson` shown below is parameterized by the number of mfds patterns to be generated. It creates four `Person` objects for the respective roles, assigns attribute values to the objects, links the generated objects in order to build a family, and finally links two generated mfds patterns by either two parenthood links or one parenthood link or no parenthood link at all. The decision is taken in a random way. For example, for a call to `genMfdsPerson(2)` a generated model could look like one of the three possibilities shown in Figure 4. Marriage links are always displayed horizontally, whereas parenthood links are shown vertically or diagonally.

```
procedure genMfdsPerson(numMFDS:Integer)       -- number of mfds patterns
 var lastNames:Sequence(String), m:Person ... -- further variables
begin
------------------------------------------------ variable initialization
lastNames:=[Sequence{'Kennedy' ... 'Obama'}];                  -- more
firstFemales:=[Sequence{'Jacqueline' ... 'Michelle'}];         -- constants
firstMales:=[Sequence{'John' ... 'Barrack'}];                  -- instead
ages:=[Sequence{30,36,42,48,54,60,66,72,78}];                  -- of ...
mums:=[Sequence{}]; dads:=[Sequence{}];

---------------------------------------------------- creation of objects
for i:Integer in [Sequence{1..numMFDS}] begin
  m:=Create(Female); f:=Create(Male);                    -- mother father
  d:=Create(Female); s:=Create(Male);                     -- daughter son
  mums:=[mums->append(m)]; dads:=[dads->append(f)];

  -- - - - - - - - - - - - - - - - - - - - - - -  assignment of attributes
  lastN:=Any([lastNames]); firstN:=Any([firstFemales]);
  [m].fullName:=[firstN.concat('␣').concat(lastN)];[m].age:=Any([ages]);
  firstN:=Any([firstMales]);
  [f].fullName:=[firstN.concat('␣').concat(lastN)];[f].age:=Any([ages]);
  ...                        -- analogous handling of daughter d and son s

  -- - - - - - - - - - - - - - - - - - - - - - - - -  creation of mfds links
  Insert(Marriage,[m],[f]);
  Insert(Parenthood,[m],[d]); Insert(Parenthood,[f],[d]);
  Insert(Parenthood,[m],[s]); Insert(Parenthood,[f],[s]);
  ---------- random generation of additional links between mfds patterns
  ------------------------------ such links lead to negative test cases
 flagA:=Any([Sequence{0,1,2,3}]); -- 0 none, 1 mother, 2 father, 3 both
  if [i>1 and flagA>0] then begin
    if [flagA=1 or flagA=3] then begin
      flagB:=Any([Sequence{0,1}]);         -- 1 give daughter, 0 give son
      if [flagB=1] then begin
        Insert(Parenthood,[mums->at(i-1)],[mums->at(i)]); end
      else begin
```

```
        Insert(Parenthood,[mums->at(i-1)],[dads->at(i)]); end;
      end; ...
end;
end;
```

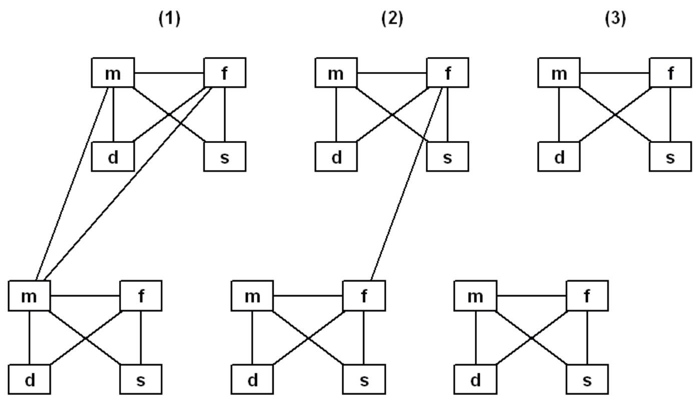


Fig. 4. Three Possibilities for Linking Two mfds Patterns

3.3 Proof of Concept

The approach we have presented in this paper allows modellers to check the behaviour of a transformation by specifying a set of tracts that should be fulfilled. For each of these tracts we generate the tract test suite mentioned in the previous section, i.e. the sample input models for the tract, and then we check that the corresponding output models (i.e., the ones produced by the transformation) fulfil the tract invariants.

As a proof-of-concept of our proposal we have built a prototype that allows testing a transformation in an automated way, chaining three tools. In the first place, the tract classes and their associated invariants are specified using USE. The ASSL program that generates the tract test suite is also specified within the USE environment, and then executed within it. The second tool is a script that takes the input models generated by the ASSL procedure (which are in the textual format that both ASSL and USE understand, `.cmd`), converts them into the Ecore format so that they can be manipulated by ATL, invokes the ATL transformation under test, and converts the resulting target model into the USE `.cmd` format again (using an ATL query). Finally, the correctness of these output models is checked against the OCL invariants specified in the transformation tract using USE.

4 Analysis

Counting on mechanisms for specifying tract invariants on the source and target metamodels, and on the relationship that should be established between them, has proved to be beneficial when combined with the testing process defined above.

Transformation Code Errors: In the first place, we can look for errors due to either bugs in the transformation code that lead to misbehaviours, or to hidden assumptions made by the developers due to some vagueness in the (verbal) specification of the transformation. These errors are normally detected by observing how valid input models (i.e., belonging to the grey area in the left hand side of Figure 1) are transformed into target models that break either the target metamodel constraints or the source-target constraints. This is the normal kind of errors pursued by most MT testing approaches.

Transformation Tract Errors: The second kind of errors can be due to the tract specifications themselves. Writing the OCL invariants that comprise a given tract can be as complex as writing the transformation code itself (sometimes even more). This is similar to what happens with the specification of the contract for a program: there are cases in which the detailed description of the expected behaviour of a program can be as complex as the program itself. However, counting on a high-level specification of what the transformation should do at the tract level (independently of how it actually implements it) becomes beneficial because both descriptions provide two complementary views (specifications) of the behaviour of the transformation. In addition, during the checking process the tract specifications and the code help testing each other. In this sense, we believe in an incremental and iterative approach to model transformation testing, where tracts are progressively specified and the transformation checked against them. The errors found during the testing process are carefully analyzed and either the tract or the transformation refined accordingly.

Issues due to Source-Target Semantic Mismatch: This process also helps revealing a third kind of issues, probably the most difficult problems to cope with. They are due neither to the transformation code nor the tract invariant specifications, but to the semantic gap between the source and target metamodels. We already mentioned that the metamodels used to illustrate our proposal look simple but hide some subtle complications. For example, one of the tracts we tried to specify was for input source models that represented three-generation families, i.e., mfds patterns linked together by parenthood relations (see Figure 5 representing a generated negative test case failing to fulfill `SRC_allPersonInMfds`; without the links `('Elizabeth Reagan', 'Ronald Reagan')`, `('Alta Reagan', 'John Carter')`, and `('Ronald Reagan', 'John Carter')` we would obtain a valid `mfds` source model). This revealed the fact that valid source models do not admit in general persons with grandchildren. More precisely, after careful examination of the problem we discovered that such patterns are valid inputs for the transformation only if the last name of all persons in the family is the same. This is because the transformed model will consist of three families, where one of the members should end up, for example, playing the role of a daughter in one family and the role

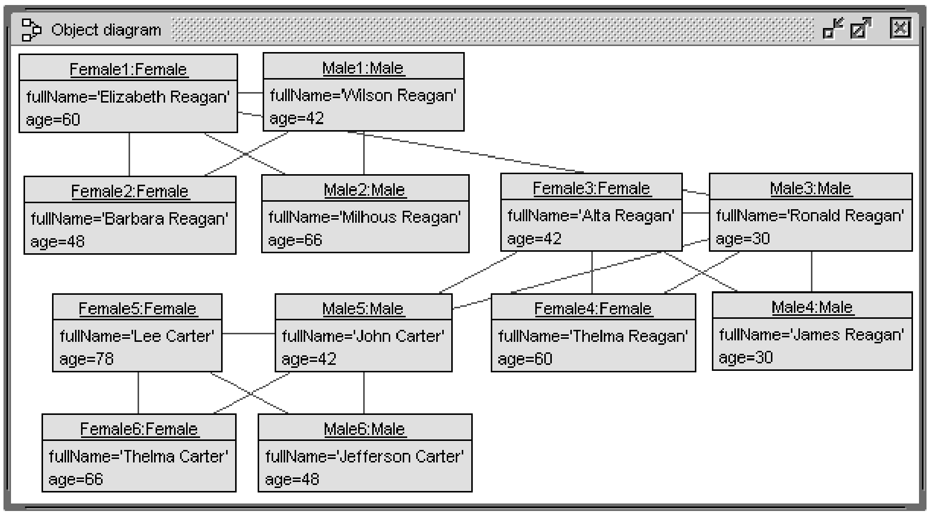


Fig. 5. Generated Negative Test Case with Linked mfds Patterns

of mother in the other. Since all members of a family should share the same last name, and due to the fact that a person should belong to two families, the last names of the two families should coincide.

Examples of these problems can also happen because of more restrictive constraints in the target metamodel. For instance a family in the target metamodel should have both a father and a mother, and they should share the same last name. This significantly restricts the set of source models that can be transformed by *any* transformation because it does not allow unmarried couples to be transformed, nor families with a single father or mother. Married couples whose members have maintained their last names cannot be transformed, either. Another problem happens with persons with only a single name (i.e., neither a first nor last name, but a name only), because they cannot be transformed. These are good examples of semantic mismatches between the two metamodels that we try to relate through the transformation. How to deal with (and solve) this latter kind of problems is out of the scope of this paper, here we are concerned only with the detection of such problems. A visual representation of some semantic differences between the example metamodels is shown in Figure 6.

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 MT tester to concentrate on specific behaviours. In this way we are able to partition the full input space of the transformation into smaller, more focused behavioural units, and to define specific tests for them. By selecting particular patterns we can traverse the full input space, checking specific spots. This is how we discovered that the size of the grey area in Figure 1 was much smaller than we initially thought, as mentioned above.

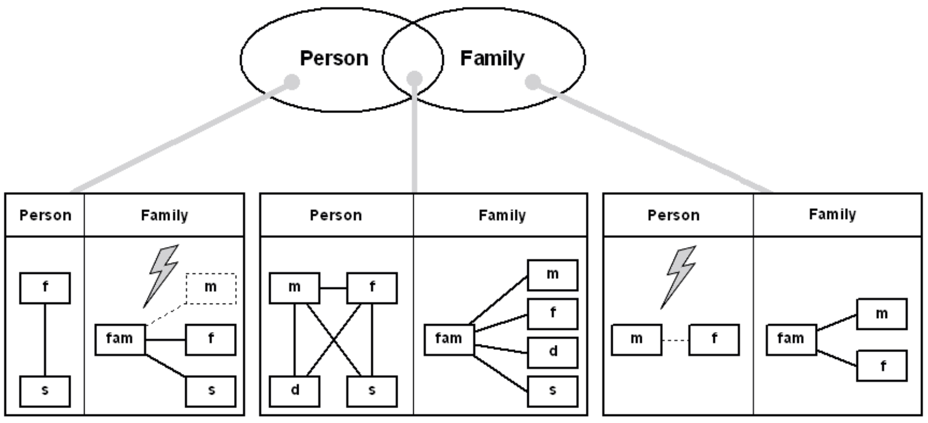


Fig. 6. Semantic Differences between Source and Target Example Metamodels

This approach also opens up the possibility of testing the transformation with invalid inputs, to check its behaviour. For example, we defined a tract where people could have two male parents, being able to check whether the transformation produced output models that violated the target metamodel constraints or not, or just hanged. In this way we can easily define both positive and negative tests for the transformation.

5 Related Work

There are several kinds of contributions that can be related to our work. In the first place we have the works that define contracts for model transformations, using different notations. One of the earlier works [18] introduces the concept of "transformation contract" in a similar way to ours—although without incorporating test suites. However, the authors propose to specify contracts by means of OCL operations, which causes many technical problems for writing contracts—as the own authors discuss in their paper. Besides, they do not discuss any practical way of using their contract specifications for model testing. The work in [2] also proposes OCL for defining transformation contracts. Their ideas are also close to [18] and to ours, but in their paper they just provide a general view of what they think that could be done with model transformation contracts, but without delving into the details about how to achieve it.

The proposal presented in [9] is also of interest. The authors show how to derive some invariant-based verification properties that should be preserved by the transformation (which are similar to our tracts) by analysing the internal rules that compose a transformation. Although they follow a white-box approach to model transformation testing, it could probably be combined with ours if their approach could help us identify some more tracts for a transformation written in any of the languages they deal with (TGG and QVT).

Other group of works (see, e.g., [5,6,7,8,10,11]) also use a white-box approach to model-transformation testing, aiming at fully validating the behaviour of the transformation (including other properties such as confluence of the rules, termination, etc.) using formal methods and their associated toolkits—which include, e.g., Alloy, Maude, or graph rewriting techniques. Although more powerful than our approach from a theoretical perspective, their computational complexity generally makes them inappropriate for testing large model transformations. In addition, the drawback of a white-box approach is that it is tightly coupled to the transformation language and thus it would need to be adapted or completely redefined for another transformation language [4].

Alternative approaches validate the transformation under test with only a selected set of test input models, as we do in our approach, but focusing more on how to generate input models [7,12,13,14,15,16] and how to reason about some of the properties of the generated set (e.g, coverage). They can be related to what we achieve with the ASSL language, although in this paper we have also shown how these input suites can be integrated into a complete testing process such as the one presented here. Complementing our current ASSL procedures with some of the ideas and algorithms proposed in these works is something we would like to explore as part of our future work.

6 Conclusions

In this paper we have introduced the concept of *Tract*, a generalization of model transformation contracts, and showed how it can be used for model transformation black-box testing. A tract defines a set of constraints on the source and target metamodels, a set of source-target constraints, and a tract test suite, i.e., a collection of source models satisfying the source constraints. To test a transformation T we automatically generate the input test suite models using the ASSL language, and then transform them into their corresponding target models. These models are checked with the USE tool against the constraints defined for the transformation. The checking process can be automated, allowing the model transformation tester to process a large number of models in a mechanical way.

Although this approach to testing does not guarantee full correctness, it provides very interesting benefits. In particular, it can be useful for identifying bugs in a cost-effective manner. Moreover, it allows dealing with industrial-size transformations without having to transform them into any other formalism or to abstract away any of its features. Furthermore, tracts provide a modular approach to testing, allowing to partition the full input space of the transformation into smaller, more focused behavioural units, and to define specific tests for them. These are important advantages over other approaches that aim at proving full correctness but at a higher-cost: their computational complexity normally make them *untractable*.

There are several lines of work that we plan to address next. In particular, we would like to study how to improve our proposal by incorporating some of the existing works on the effective generation of input test cases. We expect this to help us enhance the definition of our tract test suites. Larger case studies will be carried out in order to stress the applicability of our approach and to obtain more extensive feedback. In this sense, we would like to conduct some empirical studies on the effects of the use of tracts in the lifecycle of model transformations. Concerning the tracts, we also plan to investigate some of their properties, such as their composability, subsumption, refinement or coverage. Finally, we plan to improve the current tool support for tracts, incorporating the creation and maintenance of libraries of tracts, and the concurrent execution of the tests using sets of distributed machines.

Acknowledgements. The authors would like to thank the anonymous for their useful comments and suggestions, and to Lars Hamann, Mirco Kuhlmann, Fernando López and Javier Troya for their help and support during the preparation of the paper. This work is supported by Research Projects TIN2008-03107 and P07-TIC-03184.

References

1. Lin, Y., Zhang, J., Gray, J.: Model comparison: A key challenge for transformation testing and version control in model driven software development. In: OOPSLA/GPCE: Best Practices for Model-Driven Software Development, Control in Model Driven Software Development, pp. 219–236. Springer, Heidelberg (2004)
2. Baudry, B., Dinh-Trong, T., Mottu, J.M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Traon, Y.L.: Model transformation testing challenges. In: Proc. of the ECMDA workshop on Integration of Model Driven Development and Model Driven Testing, Barcelona, Spain (2006), `http://www.cs.colostate.edu/~france/publications/TransTesting.pdf`
3. Stevens, P.: A landscape of bidirectional model transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE II. LNCS, vol. 5235, pp. 408–424. Springer, Heidelberg (2008)
4. Baudry, B., Ghosh, S., Fleurey, F., France, R., Traon, Y.L., Mottu, J.M.: Barriers to systematic model transformation testing. Communications of the ACM 53(6), 139–143 (2010)
5. Baresi, L., Ehrig, K., Heckel, R.: Verification of model transformations: A case study with BPEL. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 183–199. Springer, Heidelberg (2007)
6. Ehrig, H., Ehrig, K., Lara, J.D., Taentzer, G., Varró, D., Varró-Gyapay, S.: Termination criteria for model transformation. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 49–63. Springer, Heidelberg (2005)
7. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Software and Systems Modeling 8, 479–500 (2009)
8. Küster, J.M.: Definition and validation of model transformations. Software and Systems Modeling 5(3), 233–259 (2006)
9. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software 83(2), 283–302 (2010)

10. Anastasakis, K., Bordbar, B., Küster, J.M.: Analysis of model transformations via Alloy. In: Proc. of MODEVVA (2007), `http://www.cs.bham.ac.uk/~bxb/Papres/Modevva07.pdf`
11. Troya, J., Vallecillo, A.: Towards a rewriting logic semantics for ATL. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 230–244. Springer, Heidelberg (2010)
12. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: Proc. of the 17th International Symposium on Software Reliability Engineering (ISSRE 2006), pp. 85–94 (2006)
13. Solberg, A., Reddy, R., Simmonds, D., France, R., Ghosh, S.: Developing distributed services using an aspect-oriented model driven framework. International Journal of Cooperative Information Systems 15(4), 535–564 (2006)
14. Mottu, J.M., Baudry, B., Traon, Y.L.: Reusable MDA components: A testing-for-trust approach. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 589–603. Springer, Heidelberg (2006)
15. Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.L.: Qualifying input test data for model transformations. Software and Systems Modeling 8(2), 185–203 (2009)
16. Küster, J.M., Abd-El-Razik, M.: Validation of model transformations – first experiences using a white box approach. In: Auletta, V. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 193–204. Springer, Heidelberg (2007)
17. Gogolla, M., Hamann, L., Kuhlmann, M.: Proving and visualizing OCL invariant independence by automatically generated test cases. In: Fraser, G., Gargantini, A. (eds.) TAP 2010. LNCS, vol. 6143, pp. 38–54. Springer, Heidelberg (2010)
18. 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), `http://web.univ-pau.fr/∼ecariou/papers/workshop-ocl-mde-uml2004-paper.pdf`
19. 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)
20. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
21. Object Management Group: Object Constraint Language (OCL) Specification. Version 2.2. (2010), OMG Document formal/2010-02-01
22. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
23. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
24. Kolovos, D.S., Rose, L.M., Paige, R.F.: The Epsilon Book (2010), `http://www.eclipse.org/gmt/epsilon/doc/book/`