Formal Specification and Testing of Model Transformations

Antonio Vallecillo[1], Martin Gogolla[2], Loli Burgueño[1], Manuel Wimmer[1], and Lars Hamann[2]

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

Abstract. In this paper we present some of the key issues involved in model transformation specification and testing, discuss and classify some of the existing approaches, and introduce the concept of *Tract*, a generalization of model transformation contracts. We show how Tracts can be used for model transformation specification and black-box testing, and the kinds of analyses they allow. Some representative examples are used to illustrate this approach.

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, which are becoming commonplace in most MDE applications, and for which manual debugging is no longer possible. Being now critical elements in the software development process, their correctness becomes essential for ensuring that the produced software applications work as expected and are free from errors and deficiencies. In particular, we do need to check whether the produced models conform to the target metamodel, or whether some essential properties are preserved by the transformation.

In general, correctness is not an absolute property. Correctness needs to be checked against a *contract*, or *specification*, which determines the expected behaviour, the context whether such a behaviour needs to be guaranteed, as well as some other properties of interest to any of the stakeholders of the system (in this case, the users of a model transformation and their implementors). A specification normally states *what* should be done, but without determining *how*. An additional benefit of some forms of specifications is that they can also be used for testing that a given implementation of the system (which describes the *how*, in a particular platform) conforms to that contract.

In general, the specification and testing of model transformations are not easy tasks and present numerous challenges [1–4]. Besides, the kinds of tests depend on the specification language and vice-versa. Thus, in the literature there are two main approaches to model transformation specification and testing (see also Section 3). 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., [5–11]). The potential limitations of these proposals lie in their inherent computational complexity, which makes them inappropriate for fully specifying and testing large and complex model transformations. An alternative approach (proposed in, e.g., [12–15, 8]) consists using declarative notations for the specification, and then 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 show a proposal that follows 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 specification and validation [16]. In particular, we generalize *model transformation contracts* [2, 17] for the specification of the properties that need to be checked for a transformation, and then apply the ASSL language [18] 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 [19].

In the following we will assume that readers are familiar with basic Software Engineering techniques such as program specification (using, in particular, pre- and postconditions [20]) and program testing (using, e.g., JUnit); with modeling techniques using UML [21] and OCL [22]; and have basic knowledge of model transformations [23].

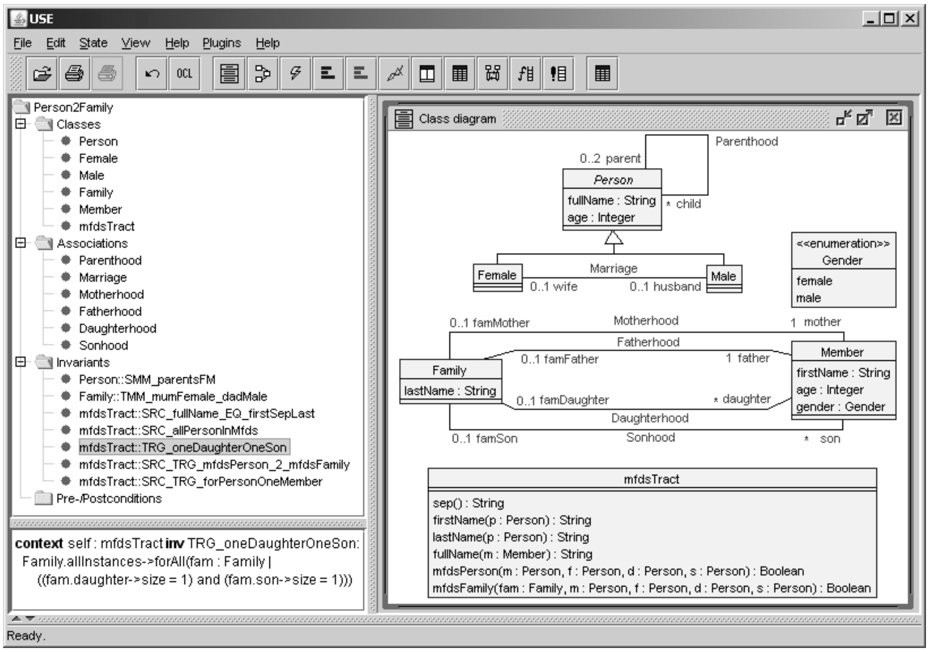
This paper is organized as follows. After this introduction, Section 2 describes the context of our work and Section 3 presents existing related works. Then, Section 4 presents our proposal and Section 5 discusses the kinds of tests and analysis that can be conducted and how to perform them. Tracts are illustrated in Section 6 with several application examples. Finally, Section 7 draws the final conclusions and outlines some future research lines.

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.

Figure 1 shows our first running example as handled by the tool USE [19]. The aim of the example is to transform a `Person` source metamodel shown in



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

Fig. 1. USE Screenshot with the `Families2Person` example

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. (This example is just the opposite to the typical `Families2Person` model transformation example [24, 25], that we shall also discuss later in Section 6.1.)

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 [26] 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`).

```
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
```

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.

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 of 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 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 one example of model transformation, we asked some students to write the model transformation that, given a `Family` model, creates a `Person` model described in the example above. The resulting code of the `Persons2Family` transformation is shown below. It is written in ATL [27], a hybrid model transformation language containing a mixture of declarative and imperative constructs which is widely used in industry and academia. There are of course other model transformation languages, such as for instance QVT [28], RubyTL [29] or JTL [30], that we could have also used. Nevertheless, in this paper we will mainly focus on ATL for illustration purposes.

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`).

```
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)) )
}
```

The question is whether this transformation is *correct*. For that we need to determine first which is expected behaviour (i.e., its specification) and then test whether the provided implementation conforms to such a specification.

3 Related Work

The need for systematic verification of model transformations has been documented by the research community by several publications outlining the challenges to be tackled [31–33, 4]. As a response, a plethora of approaches ranging from lightweight certification to full verification have been proposed to reason about different kinds of properties of model transformations [34]. Before specification and testing approaches for model transformations are discussed in more detail, the broader landscape of transformation properties is spanned first.

3.1 Categories of Model Transformation Properties

The right hand side of figure 2 (column *model transformation (MT) implementation*) aligns different kinds of properties for model transformations with the well-known model transformation pattern [35]. The transformation pattern gives an overview of the main concepts involved in model transformation. A model transformation is represented by a transformation model (*TM* in the *description* layer) has to conform to a model transformation language (described by a metamodel, *TMM* in the *language* layer) and analogously, the execution of the model transformation (*TM Ex* in the *execution* layer) has to conform to the description layer for producing from a source model (*SoM*) a corresponding target model (*TaM*).

Having this model transformation pattern as a framework for classifying model transformation properties which have been discussed in literature, the first discriminator for classifying them is the level on which they are introduced. In particular, two kinds of properties may be distinguished: (i) *general transformation properties* defined on the *language layer* allow to make statements about

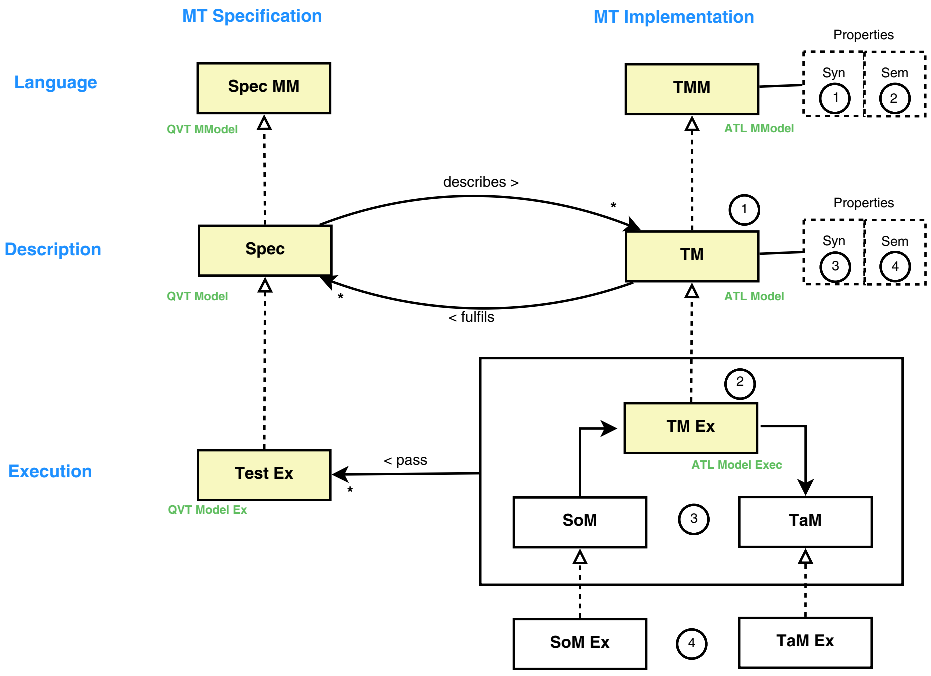


Fig. 2. Specification and testing of model transformations at a glance

transformations themselves and (ii) *specific transformation properties* defined on the *description layer* allow to make statements about pairs of *source* and *target models* of a transformation execution. Properties of the first kind abstract from the specifics of a transformation problem and are therefore usable for every transformation defined in a transformation language offering such properties. Properties of the second kind are always specific to a transformation problem, and thus, have to be defined for each transformation individually.

Orthogonal to the distinction between general and specific, is the distinction if properties are related to syntax or semantics. Thus, properties may be partitioned into *syntactic* and *semantic* properties. While syntactic properties are stated and checked based on the information provided by the next lower layer, for specifying semantic properties two steps down the stack have to be made. To be more concrete, for general transformation properties (defined on the *language* layer), the syntactic properties are calculated based on the transformation (defined on the *description* layer). However, the semantic properties have to be verified by taking the knowledge of the execution of the transformation into account (available on the *execution layer*, defined by the transformation execution engine). Analogously, for specific transformation properties, syntactical properties of the source and target model as well as their relationships are directly calculated using the models. For semantic properties, again the execution of the models has to be considered. This means, knowledge on the model execution engines used for the source and target models is needed to reason about their semantic properties.

General properties. General properties may be calculated based solely on the knowledge of the transformation language. It has to be emphasized that these properties are about the transformation, i.e., only statements about the transformation itself can be made, but not about the source and target models of the transformation.

An example for a basic general syntactic property is conformance of the transformation to the transformation language. A transformation language may be either *generic* or *specific* to a transformation problem, i.e., in the second case, the metamodels of the source and target models are considered to form an important part of the transformation language. This allows to provide enhanced syntactic checks compared to just using a generic transformation language. Approaches how to build specific transformation languages are presented in [36] for graphical modeling languages and in [37] for textual ones.

Several general semantic properties have been proposed for model transformation languages such as confluence [6, 8], applicability [38], and termination [39] of a set of graph transformation rules. Other group of works aim at fully validating the behavior of the transformation using formal methods and their associated toolkits. For example, in [11] model transformations defined in ATL are translated to Maude for analyzing them using out-of-the-box verification techniques.

Specific properties. In addition to general properties, there are properties that are specific for a certain transformation. In particular, this means that it is not enough to reason about the transformation itself: statements about the source and target models are also needed. In particular, this is a must when one has to reason about the correctness of the translation of the source model into a target model. As models comprise syntax as well as semantics, both aspects have to be considered.

Concerning syntactic properties, one may reason about if for each source element of a certain type a corresponding target element of a certain type is produced by the transformation. Such concerns are naturally formulated as contracts by using specification languages which allow to state the requirements which have to be fulfilled by a transformation implementation. Contracts [20] are a well-established technique in software engineering in general and in particular for verifying object-oriented programs by providing pre- and post-conditions as well as invariants for operations. Inspired by this work, contracts have also been applied for model transformations. In particular, as is explained in the next subsection in more detail, contracts allow for several benefits such as they can be used as oracle functions for testing model transformations by using a set of test source models. Oracle functions give an approximation of the target models which should be produced by the transformation.

For dealing with semantic properties which have to be fulfilled by the source and target models, their execution have to be taken into account. Thus, the operational semantics of the source and target languages are needed as a prerequisite.

Reasoning about semantic properties of models ranges from reasoning about some selected behavioral property such as liveness or deadlock freeness to a more complete notion of behavioral equivalence, e.g., based on bi-similarity. For example, if liveness is guaranteed by a source model, one may be interested in a transformation which generates from such models always target models guaranteing liveness as well. Furthermore, one may reason about bi-similarity of the source and target model pairs, i.e., an observer should not be able to differentiate the state-transition systems generated by the source and target models. For instance, [40] describes such an approach where each execution of the transformation is verified by checking whether the target model bi-simulates the source model. Another similar approach is presented in [41] where a model checker is used to check dynamic properties of the source and target models. It has to be noted that semantic properties are not limited to behavioral models, but may also be verified for structural models. For example, in [42] an approach is presented for reasoning about semantic differences between class diagrams by comparing all possible instantiations of them.

This chapter is dedicated to the *specification* and *testing* of *transformation specific* and *syntax related* properties of model transformations. Thus, in the following subsection, approaches going in this direction are elaborated in more detail. For a more in-depth discussion of approaches supporting the verification of other kind of properties, we kindly refer the interested reader to [34].

3.2 Specification and Testing Approaches for Model Transformations

The left hand side of figure 2 (column *model transformation (MT) specification*) focusses on the specification of specific syntactic properties and their verification. The relationships between the left hand side and the right hand side of figure 2 illustrates how transformation specifications are related to transformation implementations. As mentioned before, these properties are naturally defined in terms of contracts which form the specification for a transformation implementation. One of the advantages of contracts is that they allow defining *what* a piece of software does but not *how* it is done. In the context of model transformations, basic syntactic contracts are specified by the source and target metamodels since source and target models must conform to them. However, further restrictions on the source and target models as well as on their relationships are needed [14]. First, contracts can be used to precisely specify the constraints (going beyond metamodel constraints) to be satisfied by source models such that the transformation is applicable, i.e., *preconditions* of the transformations. Second, they can be used to express constraints on the target models, i.e., *postconditions* of the transformation. Finally, they can be used to specify constraints that need to be satisfied by any pair of source/target models of a correct transformation. Thus, a specification language should allow to formulate these three kinds of contracts.

Model transformation contracts may be used for several scenarios [17]. (i) Contracts are useful information for the transformation designer in the development and maintenance phase. (ii) They can be used to check the compatibility of transformations in a model transformation chain, e.g., the postconditions of a preceding transformation have to be compatible with the preconditions of a succeeding transformation. (iii) Contracts may be used as oracle functions to approximate the expected output for a given source model.

Especially, this latter aspect has been the subject of several kinds of works that apply contracts for model transformation testing using different notations for defining the contracts. In the following, we elaborate on these approaches which are divided into the two main categories. First, contracts may be defined on the *model level* by either giving (i) complete examples of source and target model pairs, or (ii) giving only model fragments which should be included in the produced target models for given source models. Second, contracts may be defined on the *metamodel level* either by using (iii) graph constraint languages or (iv) textual constraint languages such as OCL.

Contracts at Model Level

Model Examples. A straight-forward approach is to define the expected target model for a given source model which acts as a reference model for analyzing the actual produced target model of a transformation as proposed in [43, 1, 44, 45]. Model comparison frameworks are employed for computing a difference model between the expected and the actual target models. If there are differences then there is considered to be an error either in the transformation or in the source/target model pair. The advantage of this approach is its simplicity, e.g., as specification language, the source and target metamodels are sufficient. However, reasoning about the cause for the mismatch between the expected and actual target model solely based on the difference model is challenging. Even more aggravating, several elements in the difference model may be caused by the same error, however, the transformation engineer has the burden to cluster the differences by herself.

Fragments. A special form of verification by contract was presented in [46]. The authors propose to use model fragments (introduced in [47]) which are expected to be included in a target model which is produced from a specific source model. For verifying these properties, the model fragments are matched on the produced target model. Using fragments as contracts is different from using examples as contracts. Examples require an equivalence relationship between the expected model and actual target model, while fragments require an inclusion relationship between the expected fragments and the actual target model. As for examples, the source and target metamodels are sufficient to define the specifications; but as before, this benefit comes with the price that the contracts are described at the model level. Thus, they have to be defined for each particular test source model again and again.

Contracts at Metamodel Level

Graph constraints. In [48], the authors propose to use the graph patterns supported by the VIATRA2 tool to specify contracts for model transformations at the metamodel level. However, the patterns cannot define contracts crossing the borders of one metamodel, being therefore usable to specify pre- and postconditions, but not the relations between the source and target models.

In [49] a declarative language for the specification of visual contracts is introduced for defining pre- and post-conditions as well as invariants for model transformations. For evaluating the contracts on test models, the specifications are translated to QVT Relations which are executed in check-only mode. In particular, QVT Relations are executed before the transformation under test is executed to check the preconditions on the source models and afterwards to check relationships between the source and target models as well as postconditions on the target models.

Textual constraints. The first approach using contracts for model transformations was proposed by Cariou et al. [50, 17]. The authors suggest implementing transformations with OCL. In this way, the source metamodel classes are provided with operations, which may comprise preconditions, postconditions, and invariants. Although OCL natively supports design-by-contract, OCL is not intended to specify transformations and relationships between models. Thus, the authors propose an extension for OCL that allows defining mappings between input and output model elements.

The work in [2] also proposes OCL for defining transformation contracts. Their ideas are also close to [17], 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. A similar approach for defining contracts with OCL has been proposed in [14]. Kuester et al. [8] also proposes to use OCL for the definition of transformation constraints.

In [51, 45], the Epsilon Unit Testing Language for testing model management operations is presented. The language permits defining, as already mentioned, expected target models, but in addition, test operations where post-conditions for the target models can be specified. Giner and Pelechano [52] propose a test-driven development approach for model transformations. Test cases comprising an input model together with output fragments and OCL assertions are defined before the actual transformation implementation is developed.

Finally, formal notations to specify and test model transformations may be employed. For instance, Anastasakis et al [10] convert the model transformation under test into Alloy to perform the analysis if given assertions that have to hold for a transformation. If no target model is found by Alloy for a given source model, means that the transformation does not fulfill the assertions. Similarly, ATL transformations are translated into Maude in [11] for defining their formal semantics and for conducting different kinds of formal analyses.

4 Tracts for Model Transformations

4.1 Model Transformation Contracts

One of the problems of the previous specification approaches of Model Transformations lies on its complexity. The specifications of a model transformation can

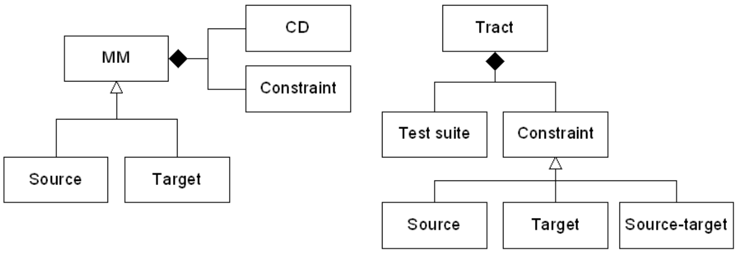


Fig. 3. Concepts in a Tract

become monstrously large as far as the transformation is not trivial (even far more complex than the transformation itself). The reasons are, among others, the lack of modularity, having to deal with too many details at the same time, and the excessive size. Because the specifications try to capture all the model transformation behaviour in one huge set of constraints, they become hard to write, debug and maintain. In addition, tests become quite cumbersome, very complex, and computationally prohibitive to prove.

In order to deal with these problems, tracts were introduced in [53] as a specification and black-box testing mechanism for model transformations. They provide modular pieces of specification, each one focusing on a particular scenario or *context of use*. Thus every model transformation can be specified by means of a set of tracts, each one covering a particular 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 allow to partition the full input space of the transformation into smaller, more focused behavioural units, and to define specific tests for them. Basically, what we do with the tracts is to identify the scenarios of interest to the user of the transformation (each one defined by a tract) and check whether the transformation behaves as expected in these scenarios. Another characteristic of our proposal is that we not require complete proofs, just to check that the transformation works for the tract test suites, hence providing a *light-weight* form of verification.

In a nutshell, 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. The constraints serve as "contracts" (in the sense of contract-based design [20]) for the transformation in some particular scenarios, and are expressed by means of OCL invariants. The provide the *specification* of the transformation. Figure 3 gives an overview on the used concepts and their connection.

Additionally, every tract provides a *test suite* that allows to operationalize the conformance tests. We do not provide the full behavioral specification of a model transformation, but just a set of tracts that defines how the transformation should behave in certain particular scenarios (or use cases) which are the ones of interest to the user. We do not care how the transformation works in the rest of the cases. In this respect, this approach is a form of *Duck typing*: "If it

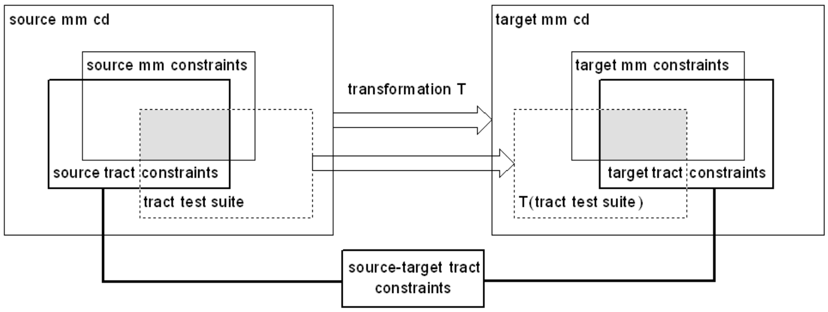


Fig. 4. Building Blocks of a Tract

looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck" [54]. Tracts are composed by conjunction, similarly to the modular specification of an operation using several pre- and post-conditions, each one defining a specific situation or use case of the operation.

In figure 4 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 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. 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.

In figure 4, 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 4 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.

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. Tracts also provide a modular approach to specification and testing, allowing to focus on particular scenarios of use, and to define precise specifications for them. These are important advantages over other approaches that prove full correctness but at a higher computational cost.

To test a transformation T against a tract t, the input test suite models can be automatically generated using languages like ASSL [18], and then transformed into their corresponding target models. These models can also be automatically checked with the USE tool [19] 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.

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 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-formulas, 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.

4.2 Generating Test Input Models

The generation of source models for testing purposes is done by means of the language ASSL (A Snapshot Sequence Language) [18]. 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 5. 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;
```

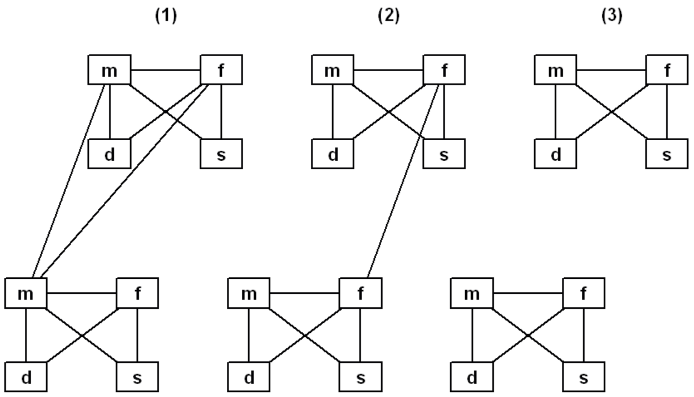


Fig. 5. Three Possibilities for Linking Two mfds Patterns

5 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

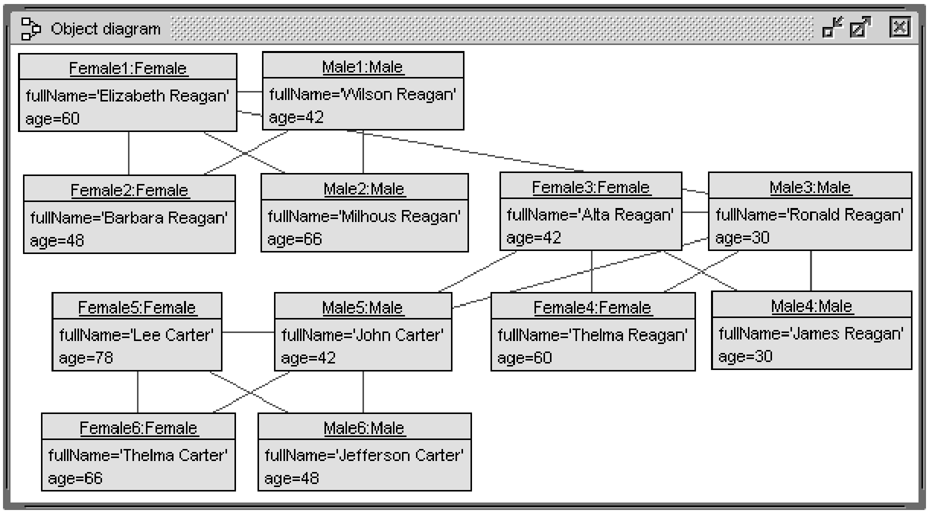


Fig. 6. Generated Negative Test Case with Linked mfds Patterns

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 6 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 of mother in the other. Since all members of a family should share the same

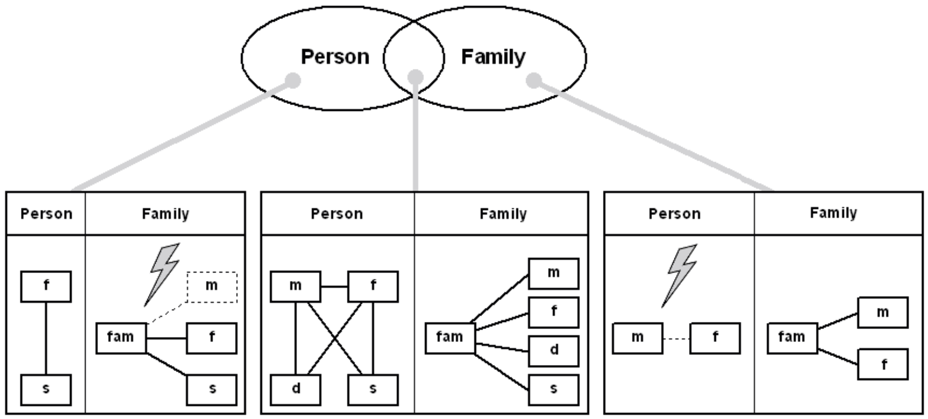


Fig. 7. Semantic Differences between Source and Target Example Metamodels

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 7.

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.

It is also worth pointing out that tracts open 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.

5.1 Model Transformation Typing Using Tracts

Tracts can also be used for "typing" model transformations. Let us explain how (sub-)typing works for tracts.

As mentioned at the beginning, what we basically do with the tracts is to identify the scenarios of interest to the user of the transformation (each one defined by a tract) and check whether the transformation behaves as expected in these scenarios. We do not care how the transformation works in the rest of the cases. This is why we consider this approach to typing is a form of "Duck" typing.

In Fig. 8 we see that `TractG` transforms metamodel `SourceG` into metamodel `TargetG`. '`G`' and '`S`' stand for 'general' (resp. 'special'). `SourceS` is a specialization of `SourceG` in the the sense that it extends `SourceG` by adding new elements (classes, attributes, associations) and possibly more restricting constraints. Analogously this is the case for `TargetS`. `TractS` is a specialization of `TractG` and inherits from `TractG` its connecting associations. Constraints must guarantee that the tract `TractS` connects `SourceS` and `TargetS` elements. Both, `TractG` and `TractS` are established with a test suite generating a set of `SourceG` models (resp. a set of `SourceS` models).

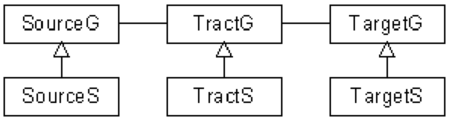


Fig. 8. Tract subtyping

In order to illustrate our typing approach, Fig. 9 shows an example for tract subtyping, using a different case study. The first source metamodel is the plain Entity-Relationship (ER) model with entities, relationships and attributes only. An ER model is identified by an object of class `ErSchema`. The second source metamodel is a specialization of the Entity-Relationship model which adds cardinality constraints for the relationship ends. Objects of class `ErSchemaC` are associated with ER models which additionally possess cardinality constraints.

The first target metamodel is the relational data model allowing primary keys to be specified for relational schemas. Objects of class `RelDBSchema` identify relational database schemas with primary keys. The second target metamodel describes relational database schemas with primary keys and additional foreign keys. The upper part of the diagram shows the principal structure with respective source and target as well as general and special elements. The lower part shows the details. Please note that the four source and target metamodels have a common part, namely the class Attribute.

It would also be possible to have disjoint source and target models by introducing classes `ErAttribute` and `ErDataType` for the ER model as well as `RelAttribute` and `RelDataType` for the relational model. The association class `ForeignKey` belongs exclusively to the relational database metamodel with foreign keys. This

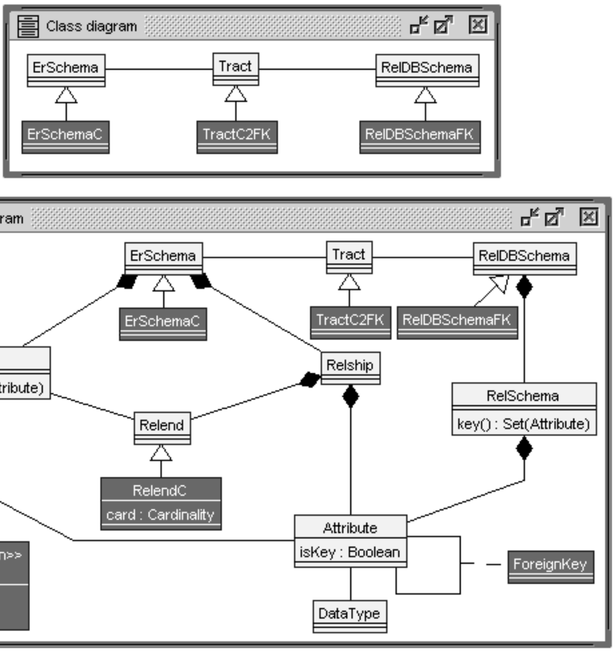


Fig. 9. An example of tract subtyping

could be made explicit by establishing a component relationship, a black diamond, from class `RelDBSchemaFK` to `ForeignKey`. The central class `Tract` specifies the transformation contract and has access, through associations, to both the source and target metamodel. Tract subtyping is expressed through the fact that class `TractC2FK` is a subtype of class `Tract`.

The scenario `Town-liesIn-Country` depicted in Fig. 10 shows informally what will be represented further down as a formal instantiation of the metamodels. Three transformations are shown. The first one `ER_2_Rel` transforms a plain ER schema (without cardinalities) into a relational database schema with primary keys only. The second one `ERC_2_Rel` goes from an ER schema with cardinalities into a relational database schema with only primary keys. The third transformation `ERC_2_RelFK` takes the ER schema with cardinalities and yields a relational database schema with primary keys and foreign keys. Please note that the three relational database schemas can be distinguished by their use of primary keys and foreign keys.

The informal scenario `Town-liesIn-Country` is formally presented in Fig. 11 with object diagrams instantiating the metamodel class diagrams. The most interesting parts which handle the primary and foreign keys are pictured in a white-on-black style. Please pay attention to the typing of the source, target, and tract objects which are different in each of the three cases and which formally reflect the chosen names of the transformations (`trafo_GG`, `trafo_SG`, `trafo_SS`).

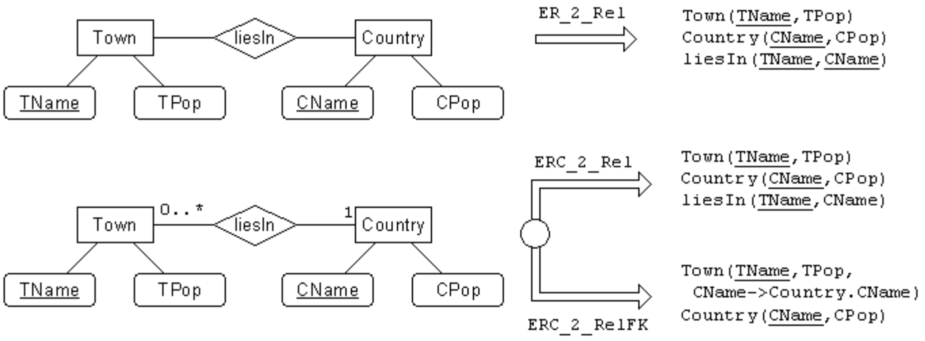


Fig. 10. `Town-liesIn-Country` scenario

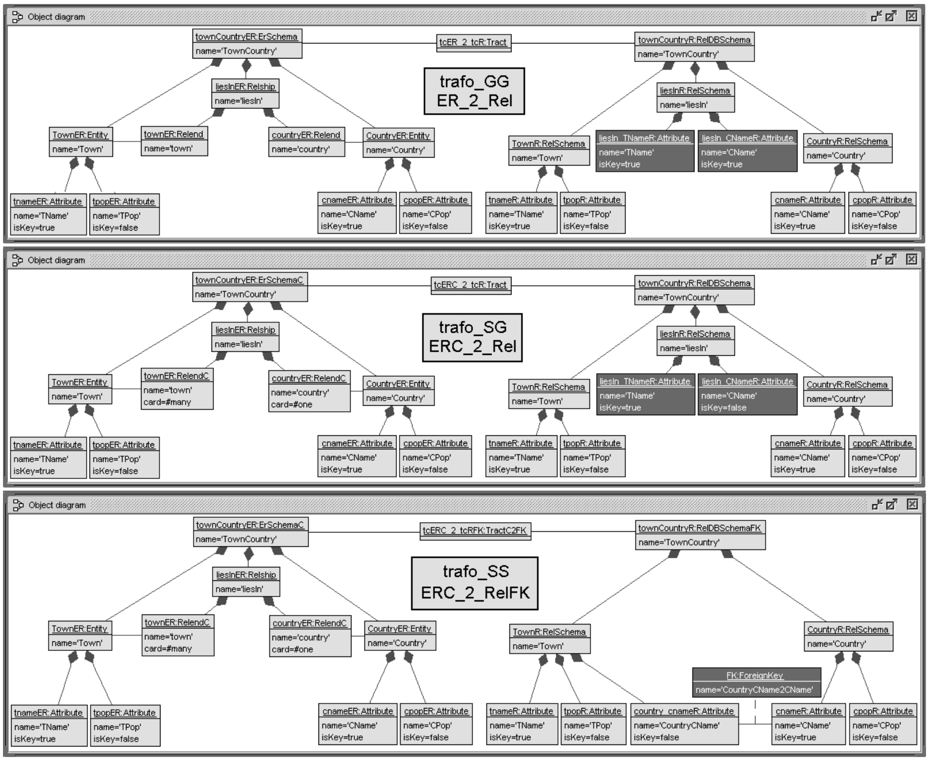


Fig. 11. `Town-liesIn-Country` object diagram

As shown in Fig. 12, in the ER and relational database metamodel example we see three different transformations: `trafo_GG`, `trafo_SS`, and `trafo_SG`. `trafo_GG` and `trafo_SS` are the transformations directly obtained from the respective tracts. Another transformation is `trafo_SG`, which takes `SourceS` models, builds `TargetG`

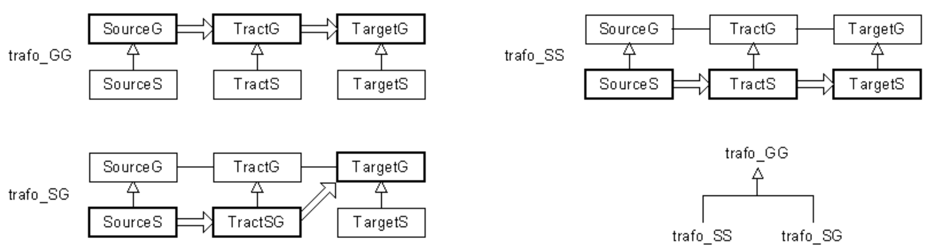


Fig. 12. Relationship between Example Transformations

models and checks them against the `TargetG` constraints. As shown in the right lower part, the example transformations `trafo_SS` and `trafo_SG` are subtypes of `trafo_GG`.

5.2 Working with Tract Types

The fact of considering tracts as types for model transformations, and the fact that tracts provide automated testing mechanisms, will allow us to perform several kinds of checks over model transformations.

Correctness of a MT Implementation. The first thing we can do is to check whether a given transformation behaves as expected, i.e., its implementation is correct w.r.t. a specification. In our approach, this is just checking that a given transformation conforms to a type. For example, a developer can come up with an ATL [27] model transformation that implements the `Families2Person` specification, and we need to test whether such MT is correct. This was the original intention of Tracts [53].

Safe Substitutability of Model Transformations. Now, given another model transformation T', how to decide whether T' can safely substitute T ($T' <: T$)? In our approach, it is a matter of testing that T' satisfies all T tracts, which can be checked in an automated way. We will not get 100% assurance that $T' <: T$ for all possible models, but we will be able to know that at least it will work in all scenarios that we have identified as relevant for us with the tracts.

Incrementality of Transformation Development. The `ERC_2_RelFK` example uses an incremental methodology for transformation development. Source and target metamodels are extended by subtyping through small increments which are accompanied by corresponding tracts including test suites. The tract test suites can give direct feedback on the correctness of the increment.

Declarative vs Imperative Tracts. Tracts may have a descriptive nature when only the relationship between source and target elements is characterized. Tracts may also be described in an operational way when the tract includes operations that map source elements to target elements. Operational tracts may be understood as implementations of descriptive ones and their correctness can be checked against the descriptive tract by employing the descriptive test suite for the operational tract.

Pros and Cons. In general, we have found that typing model transformations using tracts provides interesting advantages, such as modularity, usability, and cost-effectiveness, but at the cost of sacrificing completeness and full verification. Furthermore, having 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. Then, during the checking process the tract specifications and the code help testing each other.

5.3 Tool Support

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.

6 Further Application Examples

This sections presents two further case studies, showing their specification using tracts.

6.1 Families2Person

This section presents a set of *tracts* for the `Families2Person` model transformation, one of the simplest examples of model transformations used in the literature to explain model transformation concepts and mechanisms—it is even mentioned in the ATL documentation as some kind of ATL “hello world” example [24, 25]. Despite its apparent simplicity, the formalization of this example using tracts

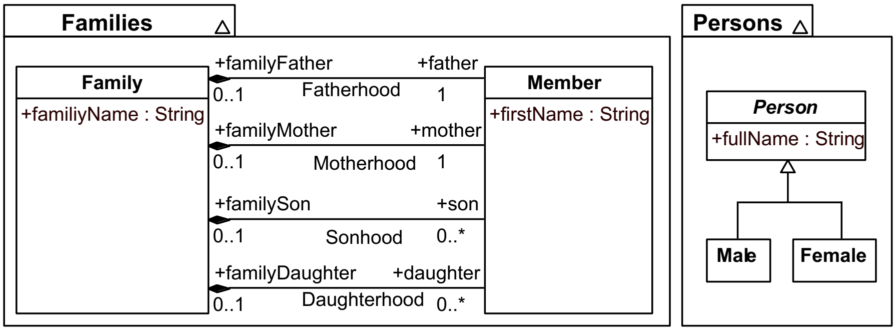


Fig. 13. `Families` and `Persons` Metamodels

has allowed us to reveal several critical problems of this transformation, which ends up being by no means *simple.*

This transformation takes models conforming to the `Families` metamodel and transforms them into models that conform to the `Persons` metamodel. Figure 13 shows these input and output metamodels. The first one describes families, which are composed of members: a father, a mother, several sons and several daughters. Each family member has a first name. In the `Persons` metamodel, a person has a full name (first name and surname), and is either a male or a female. This example follows the original specification by Freddy Allilaire and Frédéric Jouault in 2007, described in [25]. Cardinality constraints, as well as black diamonds, impose some restrictions on the relationships: for example, a family should have exactly one father and one mother. Other significant constraints are also implicitly imposed by black diamonds, as we shall later see.

The ATL transformation that implements the conversion is shown in figure 14, also taken from [25]. It has two helpers, one to decide whether a member is female or not, and other to compute the full name of members. The transformation comprises two rules, for producing male and female persons.

Tracts for the Families2Person Transformation. The `Families2Persons` transformation has been extensively used in many tutorials and papers to show a simple ATL model transformation. We therefore assume it is perfectly correct. Our aim in this section is to specify it using tracts. As mentioned earlier, tracts allow a modular specification of a model transformation whereby each tract concentrates on a set of input models (intensionally defined by the *source tract constraints*), the corresponding set of output models (intensionally defined by the *target tract constraints*), and the relationships between them as (should be) realized by the transformation (intensionally defined by the *source-target tract constraints*). In addition, every tract defines a *tract test suite* which is a collection of sample input models that are used to test the actual behaviour of the transformation.

```
module Families2Persons;
create OUT: Persons from IN: Families;

helper context Families!Member def: isFemale(): Boolean =
  if not self.familyMother.oclIsUndefined() then true
  else
     if not self.familyDaughter.oclIsUndefined() then true
     else false
     endif
  endif;

helper context Families!Member def: familyName: String =
  if not self.familyFather.oclIsUndefined() then
    self.familyFather.lastName
  else
    if not self.familyMother.oclIsUndefined() then
      self.familyMother.lastName
    else
      if not self.familySon.oclIsUndefined() then
         self.familySon.lastName
      else
         self.familyDaughter.lastName
      endif
    endif
  endif;

rule Member2Male {
  from
    s: Families!Member (not s.isFemale())
  to
    t: Persons!Male ( fullName <- s.firstName + '␣' + s.familyName )
}
rule Member2Female {
  from
    s: Families!Member (s.isFemale())
  to
    t: Persons!Female (fullName <- s.firstName + '␣' + s.familyName)
}
```

Fig. 14. ATL transformation `Families2Persons` (from [24])

Every tract is formally specified in terms of a class, that serves as context for all the OCL invariants that describe the different tract constraints.

Members Only Tract. The first tract (specified by class `MembersOnlyTract`) focuses on the simplest elements that can be used as input of the transformation: just members. According to the `Families` metamodel, a valid model may contain members associated to no family. An example of such model is shown in figure 15.

The *tract source constraint* that specifies such models is defined by OCL invariant `SCR_MembersOnly`:

```
 context MembersOnlyTract
 inv SCR_MembersOnly:
   Member.allInstances->forAll (m |
     m.familyFather->size() + m.familyMother->size() +
     m.familySon->size() + m.familyDaugther->size() = 0)
```

We need to decide what the transformation should do when these models are used as input models. In the first place, there is no restriction on the kind of persons that can be produced. So no *tract target constraint* is needed. Regarding

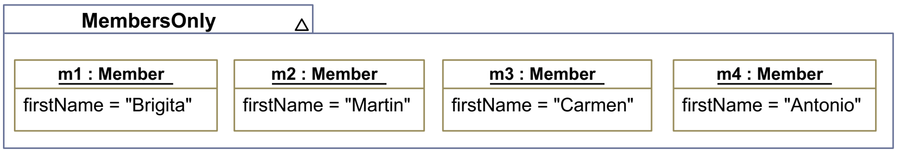


Fig. 15. A *source test model* for the `MembersOnly` tract

the *source-target constraints*, in this case there is no family to get the last name from, and there is no indication about the sex of the members. We can then decide that their full names will coincide with their first names, and that they all will be female. This is expressed by the following constraint:

```
context MembersOnlyTract
inv SRC_TRG_MembersOnly:
  Member.allInstances->forAll (m |
     Female.allInstances->one (p | p.fullName=m.firstName))
  and Member.allInstances->size() = Person.allInstances->size()
```

Now it comes to checking what the transformation does, and (with horror) we find that the transformation does not work. In fact, it aborts execution with the following error message:

```
An internal error occurred during: "Launching Families2Persons".
java.lang.ClassCastException:
  org.eclipse.m2m.atl.engine.emfvm.lib.OclUndefined cannot be cast to
    org.eclipse.m2m.atl.engine.emfvm.lib.HasFields
```

After investigating, it is due to the fact that the `familyName` attribute of variable `s` in the transformation rule is not defined. And what is worse, even if the transformation did not abort its execution, we realized that it would convert all members into male persons. And then, it would add a blank space to their names. So the exemplar transformation have not even passed our most simple test... What is wrong with all this?

In the first place, our decisions above may seem arbitrary. Why should they all become female persons and not male? In fact, it may be not fair to make any decision at all, it really does not make any sense to have no families in the `Families` model, only members. So the best option in this case is to rule out the possibility of having members with no associated families in any valid `Families` model. This is expressed by OCL constraint `NoIsolatedMembers` that provides an invariant for class `Member` in the `Families` metamodel:

```
context Member
inv NoIsolatedMembers:
  Member.allInstances->forAll (m |
    m.familyFather->size() + m.familyMother->size() +
    m.familySon->size() + m.familyDaugther->size() > 0)
```

From this moment on, we will suppose that this constraint forms an integral part of the `Families` metamodel.

No Children Tract. The second tract (specified by class `NoChildrenTract`) focuses on simple families composed of two members: a father and a mother.

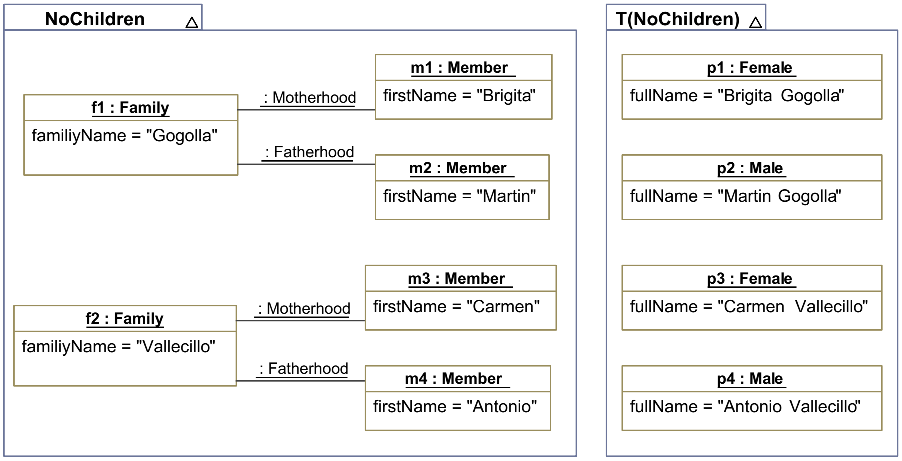


Fig. 16. Test model for the `NoChildren` tract and its corresponding transformed model

An example of such model is shown on the left hand side of figure 16. The *tract source constraint* that specifies such models is defined by OCL invariant `SCR_NoChildren`:

```
context NoChildrenTract
inv SCR_NoChildren:
  Family.allInstances->forAll(f | f.son->size()+f.daughter->size() = 0)
```

We need to decide what the transformation should do when these models are used as input models. In the first place, there is no restriction on the kind of persons that can be produced. So no *tract target constraint* is needed.

Regarding the *source-target constraints*, in this case we need to check that for every member there is one person that is either male or female depending on the role he or she plays in the family, and whose full name corresponds to the first name of the member and the family name of the family. This is expressed by the following constraint:

```
context NoChildrenTract inv SRC_TRG_NoChildren:
 Member.allInstances->forAll (m |
  m.familyMother->size()=1 implies Female.allInstances->exists(p |
  p.fullName=m.firstName.concat('␣').concat(m.familyMother.familyName)))
 and
 Member.allInstances->forAll (m |
  m.familyFather->size()=1 implies Male.allInstances->exists(p |
  p.fullName=m.firstName.concat('␣').concat(m.familyFather.familyName)))
 and
  Member.allInstances->size() = Person.allInstances->size()
```

The test suite for this tract is defined by the following ASSL procedure, which generates sample input models that conform to the `Families` metamodel to be transformed by the transformation.

```
procedure mkSourceNoChildren(numFamily:Integer, numMember:Integer,
    numMother:Integer, numFather:Integer)
  var theFamilies: Sequence(Family), theMember: Sequence(Member),
        f: Family, m: Member;
begin
  theFamilies:=CreateN(Family,[numFamily]);
  theMember:=CreateN(Member,[numMember]);
  for i:Integer in [Sequence{1..numFamily}] begin
    [theFamilies->at(i)].familyName:=Any([Sequence{'Red','Green',
            'Blue','Black','White','Brown','Amber','Yellow'}]);
  end;
  for i:Integer in [Sequence{1..numMember}] begin
    [theMember->at(i)].firstName:=Any([Sequence{
      'Ada','Bel','Cam','Day','Eva','Flo','Gen','Hao','Ina','Jen',
      'Ali','Bob','Cyd','Dan','Eli','Fox','Gil','Hal','Ike','Jan'}]);
  end;
  for i:Integer in [Sequence{1..numMother}] begin
    f:=Try([theFamilies->select(f|f.noMother())]);
    m:=Try([theMember->select(m|m.noFamily())]);
    Insert(Motherhood,[f],[m]);
  end;
  for i:Integer in [Sequence{1..numFather}] begin
    f:=Try([theFamilies->select(f|f.noFather())]);
    m:=Try([theMember->select(m|m.noFamily())]);
    Insert(Fatherhood,[f],[m]);
  end; end;
```

The following Ecore model shows an example of the models constructed in this way (also shown in figure 16), ready to serve as input to the model transformation under study:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
         xmlns="Families">
  <Family lastName="Gogolla">
    <mother firstName="Brigita"/>
    <father firstName="Martin"/>
  </Family>
  <Family lastName="Vallecillo">
    <mother firstName="Carmen"/>
    <father firstName="Antonio"/>
  </Family>
</xmi:XMI>
```

MFDS Tract. This tract (specified by class `MFDS`) focuses on families composed of exactly four members: one father, one mother, one son and one daughter. An example of such model is shown in figure 17. The *tract constraint* that specifies such models is defined by the next OCL invariants:

```
context MFDS inv SRC_OneDaughterOneSon:
  Family.allInstances->forAll(f|f.daughter->size=1 and f.son->size()=1)
context MFDS inv SRC_TRG_MotherDaughter2Female:
  Family.allInstances->forAll(fam|Female.allInstances->exists(m|
    fam.mother.firstName.concat('␣').concat(fam.familyName)=m.fullName))
  and
  Family.allInstances->forAll(fam|Female.allInstances->exists(d|
    fam.daughter->any(true).firstName.concat('␣').concat(fam.familyName)
        =d.fullName))

context MFDS inv SRC_TRG_FatherSon2Male:
  Family.allInstances->forAll(fam|Male.allInstances->exists(f|
    fam.father.firstName.concat('␣').concat(fam.familyName)
        =f.fullName))
    and
    Family.allInstances->forAll(fam|Male.allInstances->exists(s|
      fam.son->any(true).firstName.concat('␣').concat(fam.familyName)=
          s.fullName))

context MFDS inv SRC_TRG_Female2MotherDaughter:
  Female.allInstances->forAll(f|Family.allInstances->exists(fam|
    fam.mother.firstName.concat('␣').concat(fam.familyName)=f.fullName
    or
    fam.daughter->any(true).firstName.concat('␣').concat(fam.familyName)
        =f.fullName))

context MFDS inv SRC_TRG_Male2FatherSon:
  Male.allInstances->forAll(m|Family.allInstances->exists(fam|
      fam.father.firstName.concat('␣').concat(fam.familyName)=m.fullName
      or
      fam.son->any(true).firstName.concat('␣').concat(fam.familyName)
          =m.fullName))

context MFDS inv SRC_TRG_MemberSize_EQ_PersonSize:
  Member.allInstances->size=Person.allInstances->size
```

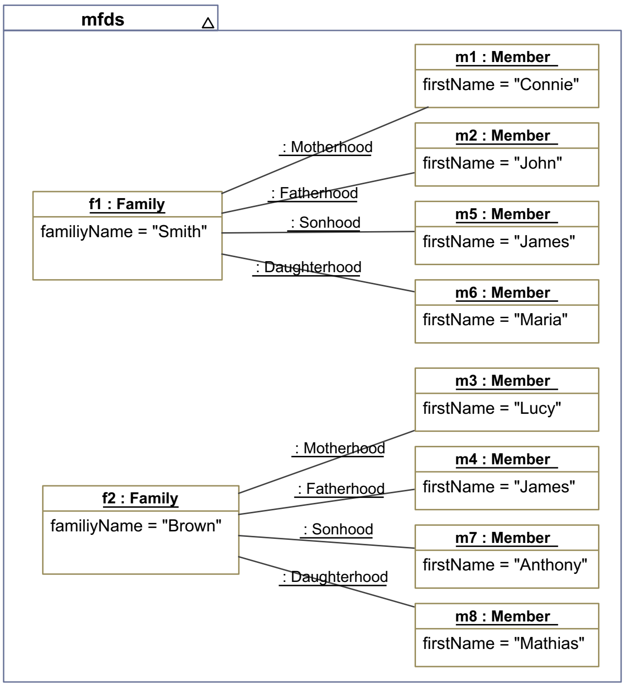


Fig. 17. A test model for the MFDS tract

And the ASSL code is:

```
procedure mkSourceMFDS(numFamily:Integer, numMember:Integer, numMother:
    Integer, numFather:Integer, numDaughter:Integer, numSon:Integer)
    var theFamilies: Sequence(Family), theMember: Sequence(Member),
        f: Family, m: Member;
begin
    theFamilies:=CreateN(Family,[numFamily]);
    theMember:=CreateN(Member,[numMember]);
    for i:Integer in [Sequence{1..numFamily}] begin
        [theFamilies->at(i)].familyName:=Any([Sequence{'Red','Green',
            'Blue','Black','White','Brown','Amber','Yellow'}]);
    end;
    for i:Integer in [Sequence{1..numMember}] begin
      [theMember->at(i)].firstName:=Any([Sequence{
        'Ada','Bel','Cam','Day','Eva','Flo','Gen','Hao','Ina','Jen',
        'Ali','Bob','Cyd','Dan','Eli','Fox','Gil','Hal','Ike','Jan'}]);
    end;
    for i:Integer in [Sequence{1..numMother}] begin
        f:=Try([theFamilies->select(f|f.noMother())]);
        m:=Try([theMember->select(m|m.noFamily())]);
        Insert(Motherhood,[f],[m]);
    end;
    for i:Integer in [Sequence{1..numFather}] begin
        f:=Try([theFamilies->select(f|f.noFather())]);
        m:=Try([theMember->select(m|m.noFamily())]);
        Insert(Fatherhood,[f],[m]);
    end;
    for i:Integer in [Sequence{1..numDaughter}] begin
        f:=Try([Family.allInstances->sortedBy(f|f.daughter->size()+
            f.son->size)->asSequence()]);
        m:=Try([theMember->select(m|m.noFamily())]);
        Insert(Daughterhood,[f],[m]);
    end;
    for i:Integer in [Sequence{1..numSon}] begin
        f:=Try([Family.allInstances->sortedBy(f|f.daughter->size+
            f.son->size())->asSequence()]);
        m:=Try([theMember->select(m|m.noFamily())]);
        Insert(Sonhood,[f],[m]);
    end; end;
```

Note that this code subsumes the previous one used in the `NoChildren` tract, which can be expressed as `mkSourceMFDS(x, y, z, 0, 0)`.

Two Generation Families Tract. Another interesting situation that we may think of happens with two-generation families, where a son or a daughter plays the role of father or mother in another. Figure 18 shows an example of this kind of input models, which represent a common case in real-world families.

The issue with this kind of families is that they are not valid models according to the `Families` metamodel, the problem being that the relationship between a family and its members is a composition (black diamond). This means that a member can belong to at most one family, i.e., a member can only play a role in at most one family.

Here, the problem is due to the source metamodel, which is too restrictive and does not allow this kind of families. A possible solution would be to change the source metamodel, relaxing it, but this is outside our hands. We wanted to respect the source and target metamodels, as well as the transformation itself, as much as possible. At most we could add some constraints if the transformation is ill-defined, to avoid problematic source models (as we have done with

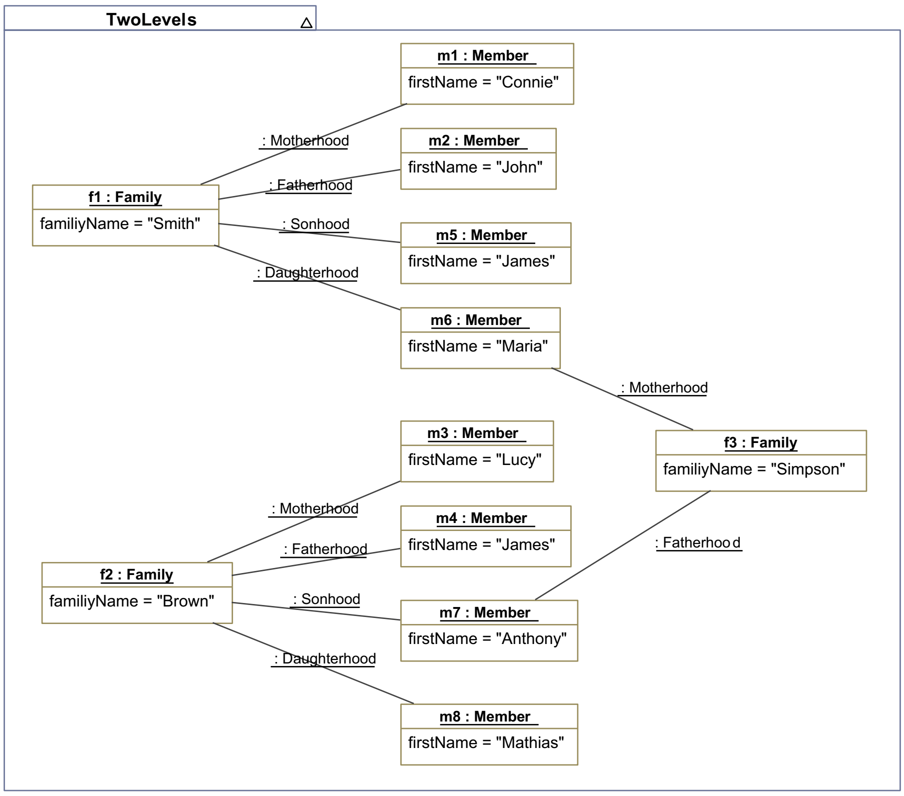


Fig. 18. A two-generations model

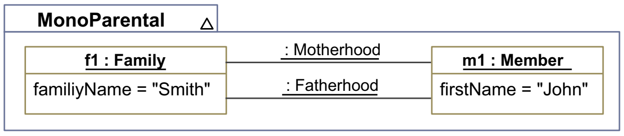


Fig. 19. A test model for the `Monoparental` tract

the first tract). But relaxing the original constraints may not be appropriate. Besides, making this kind of changes to any of the metamodels can produce undesirable effects, because the developers of the example may have made use of these constraints.

Monoparental Tract. Although a composition relation forbids an element to play two roles in two different containers, it is not so clear whether a contained element can play two different roles in the same container or not. An example in our domain would be a mono-parental family, with a person fulfilling the roles of father and mother (see figure 19).

Although UML allows this model to be drawn, and it is valid according to the UML metamodel, it is invalid both in Ecore and in USE. They do not allow the same object to play two contained roles even within the same container.

And even if this model was valid, the expected result of the transformation is not clear. The problem here is determining the sex of the person because now it cannot be inferred.

To avoid this case we suggest to add an additional constraint to the `Family` metamodel, which makes this restriction clear, and not implicit (and therefore valid in some technical spaces, and invalid in others):

```
context Member inv BlackDiamonds:
 familyMother->size() + familyFather->size() +
 familyDaughter->size() + familySon->size() <= 1
```

Note how this invariant, together with the `NoIsolatedMembers` invariant, forces the number of roles that a member can have in a family to be exactly 1.

Other Tracts. So far tracts have allowed us to identify interesting sample models for the transformation, some of which revealed problematic source models and erroneous behaviours. Others, such as the `MFDS` or `NoChildren` tracts, determined valid use cases for which the transformation should work as well as the specification of such behaviour. Other tracts for the transformation may include:

– `OnlyGirls` where families only have daughters, no sons.
– `OnlyBoys` where families only have sons, no daughthers.
– `NoFather3Kids` where families have a mother but no father, and 3 children.
– `NoMother3Kids` where families have a father but no mother, and 3 children.
– `NoFatherNoKids` where families have a mother but no father, and no children.
– `NoMotherNoKids` where families have a father but no mother, and no children.
– `OrphanSons` where families have only sons (no father, mother or daughthers).
– `OrphanDaughthers` where families have only daughters (no father, mother or sons).

The specification of these tracts are left as exercise to the reader.

Summary. Here we have illustrated the *Tract* concept with the example of the `Families2Person` transformation. It has allowed us to discover that even such a simple example is by no means trivial. In particular, the specification of this transformation using *Tracts* has uncovered one case where the transformation fails (`MembersOnly`), and has also allowed us to explore the input and output spaces of the transformation, discovering that its scope is much more reduced that we intuitively thought: only individual families are allowed, with no shared members. This rules out, for instance, most common cases of two-generation families (grandparent-parent-children) or families whose children marry members of other families. We have also discovered that the implicit restrictions imposed by some modeling constructs, such as black diamonds, are treated differently in several modeling tools. In these cases, the explicit expression of these constraints may be helpful.

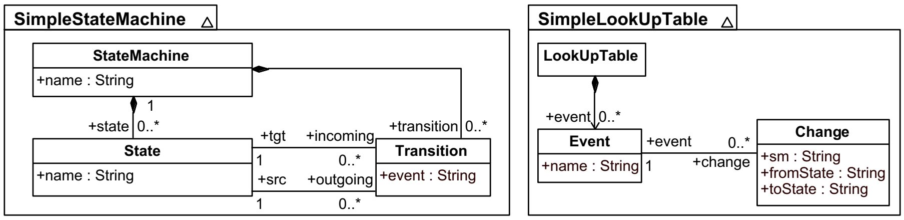


Fig. 20. Source and Target Metamodels of transformation SM2T

6.2 StateMachineTo LookUp Tables

As a second example, let us consider a model transformation SM2T between simple state machines and a lookup table that lists the events and their associated transitions [55]. The source and target metamodels of this transformation are shown in figure 20. In this case, we want only one lookup table to be built, whose entries are all the events of all the state machines in the source model. In addition to the (multiplicity) constraints shown in these class diagrams, we need to add uniqueness on names of the state machines, and uniqueness of names of states within the same state machine:

```
context StateMachine inv uniqueNames:
   self.state->isUnique(name) and
   StateMachine.allInstances->isUnique(name)
```

To specify the SM2T transformation we can define the following six tracts, whose test suite models are illustrated in figure 21 (literals SM1...SM6 represent the names of the state machines):

- 1S0T: state machines with single states and no transitions.
- 2S1T: state machines with two states and one transition between them. In this case the entries of the resulting lookup table will have the form $\{\mathtt{x} \mapsto (\mathtt{SM2}, \mathtt{A}, \mathtt{B})\}$.
- 2S2T: state machines with two states and two transition between them. In this case the entries of the resulting lookup table will be of the form $\{\mathtt{x} \mapsto (\mathtt{SM3}, \mathtt{A}, \mathtt{B}), \mathtt{y} \mapsto (\mathtt{SM3}, \mathtt{B}, \mathtt{A})\}$.
- 1S1T: state machines with single states and one transition. In this case the entries of the resulting lookup table will have the form $\{\mathtt{x} \mapsto (\mathtt{SM4}, \mathtt{A}, \mathtt{A})\}$.
- 3S3T: state machines with three states and three transitions, forming a cycle. In this case the entries of the resulting lookup table will be of the form $\{\mathtt{x} \mapsto (\mathtt{SM5}, \mathtt{A}, \mathtt{B}), \mathtt{y} \mapsto (\mathtt{SM5}, \mathtt{B}, \mathtt{C}), \mathtt{z} \mapsto (\mathtt{SM5}, \mathtt{C}, \mathtt{A})\}$.
- 3S9T: state machines with three states and 9 transitions (see figure 21). In this case the entries of the resulting lookup table will have the form $\{\mathtt{x0} \mapsto (\mathtt{SM6}, \mathtt{A}, \mathtt{A}), \mathtt{x1} \mapsto (\mathtt{SM6}, \mathtt{A}, \mathtt{B}), \mathtt{x2} \mapsto (\mathtt{SM6}, \mathtt{B}, \mathtt{A}), \mathtt{y0} \mapsto (\mathtt{SM6}, \mathtt{B}, \mathtt{B}), \mathtt{y1} \mapsto (\mathtt{SM6}, \mathtt{B}, \mathtt{C}), \mathtt{y2} \mapsto (\mathtt{SM6}, \mathtt{C}, \mathtt{B}), \mathtt{z0} \mapsto (\mathtt{SM6}, \mathtt{C}, \mathtt{C}), \mathtt{z1} \mapsto (\mathtt{SM6}, \mathtt{C}, \mathtt{A}), \mathtt{z2} \mapsto (\mathtt{SM6}, \mathtt{A}, \mathtt{C})\}$.

Let us show here one of this tracts, 2S1T, for illustration purposes. The rest follow similar patters. In the first place, the *tract source constraint* that specifies the source models is defined by OCL invariant SCR_2S1T:

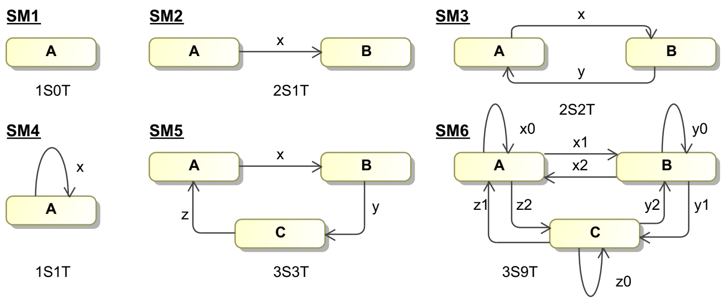


Fig. 21. Test suites samples for the 6 tracts defined for model transformation SM2T

```
context 2S1T−Tract
inv SCR_2S1T:
  StateMachine.allInstances−>forAll (sm |
      (sm.state−>size() = 2)  and (sm.transition−>size() = 1)
      (sm.transition.src <> sm.transition.tgt)
```

We need to decide what the transformation should do when these models are used as input models. There is no restriction on the kinds of entries that can be produced in the lookup table, but we need to state that only one lookup table is produced. This is expressed by the following OCL constraint:

```
context 2S1T−Tract
inv TRG_2S1T: LookUpTable.allInstances−>size() = 1
```

Regarding the *source-target constraints*, given that every state machine has only one transition, there should be one change in the lookup table for every state machine, and the attributes should match with the events and states related by the corresponding transition in the state machine. This is expressed by the following source-target constraint:

```
context 2S1T−Tract
inv SRC_TRG_2S1T:
  StateMachine.allInstances−>size() = LookUpTable.change−>size() and
  LookUpTable.change−>forAll (c |
     StateMachine.allInstances−>one(sm | (sm.name = c.sm) and
          (sm.transition.src−>collect(name) = c.fromState.asSet()) and
          (sm.transition.tgt−>collect(name) = c.toState.asSet()) and
          (sm.transition.event = c.event.name) )
```

Finally, the test suite for this tract is defined by an ASSL procedure that generates the input models.

```
procedure mk2S1T(numSM:Integer)
  var theStateMachines: Sequence(StateMachine),
    theStates: Sequence(State),
    theTransitions: Sequence(Transition);
begin
  theStateMachines:=CreateN(StateMachine,[numSM]);
  theStates:=CreateN(State,[2*numSM]);
  theTransitions:=CreateN(Transition,[numSM]);
  for i:Integer in [Sequence{1..numSM}] begin
    [theStateMachines−>at(i)].name:= ['SM'.concat(i.toString())];
        [theTransitions−>at(i)].event:= ['E'.concat(i.toString())];
        [theStates−>at(2*i−1)].name:= ['ST'.concat((2*i−1).toString())];
        [theStates−>at(2*i)].name:= ['ST'.concat((2*i).toString())];
```

```
         Insert(States,[theStateMachines->at(i)],[theStates->at(2*i-1)]);
         Insert(States,[theStateMachines->at(i)],[theStates->at(2*i)]);
   Insert(Transition,[theStateMachines->at(i)],[theTransitions->at(i)]);
   Insert(Cause,[theTransitions->at(i)],[theStates->at(2*i-1)]);
        Insert(Effect,[theTransitions->at(i)],[theStates->at(2*i)]);
   end;
end;
```

In the example above, the type of the `SM2T` transformation is given by the six tracts defined for it: `SM2T` $\models$ `1S0T` $\wedge$ `2S1T` $\wedge$ `2S2T` $\wedge$ `1S1T` $\wedge$ `3S3T` $\wedge$ `3S9T`. Of course, other tracts could have been defined for this transformation if the user requires to include further contexts of use.

7 Conclusion

In this paper we have presented the issues involved in model transformation specification and testing, and introduced the concept of *Tract*, a generalization of model transformation contracts. We have showed how it can be used for model transformation specification and 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 advantages over other approaches, as we have discussed above.

There are other issues that we have not covered in this paper, such as the generation of source models (test suites) to optimize metamodel coverage or transformation code coverage (in case of white-box testing). In this respect, 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. We would also 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 volume editors, Marco Bernardo, Vittorio Cortellessa and Alfonso Pierantonio for their invitation to present our ideas on model transformation specification and testing, and to 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 TIN2011-23795 and by the Austrian Science Fund (FWF) under grant J 3159-N23.

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: Control in Model Driven Software Development. OOPSLA/GPCE: Best Practices for Model-Driven Software Development, pp. 219–236. Springer (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 IMDD-MDT 2006 (2006)
3. Stevens, P.: A Landscape of Bidirectional Model Transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2008. 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., de Lara, J., 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.: A rewriting logic semantics for ATL. Journal of Object Technology 10(5), 1–29 (2011)
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 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., Le Traon, Y.: 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. 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)
17. 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)
18. 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)
19. 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)
20. Meyer, B.: Applying design by contract. IEEE Computer 25(10), 40–51 (1992)
21. Andova, S., van den Brand, M.G.J., Engelen, L.J.P., Verhoeff, T.: MDE Basics with a DSL Focus. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) SFM 2012. LNCS, vol. 7320, pp. 21–57. Springer, Heidelberg (2012)
22. Cabot, J., Gogolla, M.: Object Constraint Language (OCL): A Definitive Guide. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) SFM 2012. LNCS, vol. 7320, pp. 58–90. Springer, Heidelberg (2012)
23. Di Ruscio, D., Eramo, R., Pierantonio, A.: Model Transformations. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) SFM 2012. LNCS, vol. 7320, pp. 91–136. Springer, Heidelberg (2012)
24. Eclipse: ATL Tutorials – A simple ATL transformation (2007), `http://wiki.eclipse.org/ATL/Tutorials_Create_a_simple_ATL_transformation`
25. Eclipse: Basic ATL examples (2007), `http://www.eclipse.org/m2m/atl/basicExamples_Patterns/`
26. Object Management Group: Object Constraint Language (OCL) Specification. Version 2.2. OMG Document formal/2010-02-01 (2010)
27. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
28. OMG: MOF QVT Final Adopted Specification. Object Management Group. OMG doc. ptc/05-11-01 (2005)
29. Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: RubyTL: A Practical, Extensible Transformation Language. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 158–172. Springer, Heidelberg (2006), `http://rubytl.rubyforge.org/`
30. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: JTL: A Bidirectional and Change Propagating Transformation Language. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 183–202. Springer, Heidelberg (2011), `http://jtl.di.univaq.it/`
31. Baudry, B., Dinh-Trong, T., Mottu, J., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y.: Model transformation testing challenges. In: ECMDA Workshop on Integration of MDD and Model Driven Testing (2006)
32. France, R.B., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Proc. of ISCE 2007, pp. 37–54 (2007)
33. Van Der Straeten, R., Mens, T., Van Baelen, S.: Challenges in Model-Driven Software Engineering. In: Chaudron, M.R.V. (ed.) MoDELS 2008. LNCS, vol. 5421, pp. 35–47. Springer, Heidelberg (2009)

34. Amrani, M., Lúcio, L., Selim, G., Combemale, B., Dingel, J., Vangheluwe, H., Traon, Y.L., Cordy, J.R.: A tridimensional approach for studying the formal verification of model transformations. In: Proc. of the 1st International Workshop on Verification and Validation of Model Transformations, VOLT 2012 (2012)
35. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–646 (2006)
36. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Systematic transformation development. ECEASST 21 (2009)
37. Weisemoeller, I., Rumpe, B.: A domain specific transformation language. In: Models and Evolution Workshop @ MoDELS 2011 (2011)
38. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: An Invariant-Based Method for the Analysis of Declarative Model-to-Model Transformations. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 37–52. Springer, Heidelberg (2008)
39. Varró, D., Varró–Gyapay, S., Ehrig, H., Prange, U., Taentzer, G.: Termination Analysis of Model Transformations by Petri Nets. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 260–274. Springer, Heidelberg (2006)
40. Narayanan, A., Karsai, G.: Towards verifying model transformations. Electr. Notes Theor. Comput. Sci. 211, 191–200 (2008)
41. Varró, D.: Automated formal verification of visual modeling languages by model checking. Software and System Modeling 3(2), 85–113 (2004)
42. Maoz, S., Ringert, J.O., Rumpe, B.: CDDiff: Semantic Differencing for Class Diagrams. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 230–254. Springer, Heidelberg (2011)
43. Kolovos, D.S., Paige, R.F., Polack, F.A.: Model comparison: a foundation for model composition and model transformation testing. In: GaMMa 2006, pp. 13–20. ACM (2006)
44. Lin, Y., Zhang, J., Gray, J.: A testing framework for model transformations. Model-Driven Software Development, 219–236 (2005)
45. 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: Whittle, J., Clark, T., Kühne, T. (eds.) MoDELS 2011. LNCS, vol. 6981, pp. 395–409. Springer, Heidelberg (2011)
46. Mottu, J.M., Baudry, B., Traon, Y.L.: Model transformation testing: oracle issue. In: ICSTW 2008, pp. 105–112. IEEE (2008)
47. Ramos, R., Barais, O., Jézéquel, J.-M.: Matching Model-Snippets. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MoDELS 2007. LNCS, vol. 4735, pp. 121–135. Springer, Heidelberg (2007)
48. Balogh, A., Bergmann, G., Csertán, G., Gönczy, L., Horváth, Á., Majzik, I., Pataricza, A., Polgár, B., Ráth, I., Varró, D., Varró, G.: Workflow-Driven Tool Integration Using Model Transformations. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 224–248. Springer, Heidelberg (2010)
49. 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. (accepted for publication) (2012)
50. Cariou, E., Belloir, N., Barbier, F., Djemam, N.: OCL contracts for the verification of model transformations. ECEASST 24 (2009)
51. Kolovos, D., Paige, R., Rose, L., Polack, F.: Unit testing model management operations. In: ICSTW 2008, pp. 97–104. IEEE (2008)

52. Giner, P., Pelechano, V.: Test-Driven Development of Model Transformations. In: Schürr, A., Selic, B. (eds.) MoDELS 2009. LNCS, vol. 5795, pp. 748–752. Springer, Heidelberg (2009)
53. Gogolla, M., Vallecillo, A.: *Tract*able Model Transformation Testing. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 221–235. Springer, Heidelberg (2011)
54. Heim, M.: Exploring Indiana Highways: Trip Trivia. Exploring America's Highway. Travel Organization Network (2007), `http://en.wikipedia.org/wiki/Duck_test`
55. Steel, J., Jézéquel, J.M.: On model typing. Software and Systems Modeling 6(4), 401–413 (2007)