Typing Model Transformations Using Tracts

Antonio Vallecillo[1] and Martin Gogolla[2]

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

Abstract. As the complexity of MDE artefacts grows, there is an increasing need to rely on precise and abstract mechanisms that allow system architects to reason about the systems they design, and to test their individual components. In particular, assigning types to models and model transformations is needed for realizing many key MDE activities. This paper presents a light-weight approach to type model transformations using *tracts*. Advantages and limitations of the proposal are discussed, as well as the applicability of the proposal in several settings.

1 Introduction

Types are essential in Model-Driven Engineering (MDE) for understanding, managing and manipulating all artefacts involved in the analysis, design, development, operation and evolution of software systems. In particular, assigning types to models and to model transformations (i.e., *typing* them) is required for characterizing, in a precise and abstract manner, the operations we can perform on them, their valid inputs and outputs, and how they behave. Types are also very useful for ensuring their error-free composition, their safe replaceability by newer versions or by other artefacts, and for checking that a given instance or implementation is correct—by checking that it conforms to the appropriate type.

Typing models is something that the MDE community has already addressed. In a nutshell, the type of a model is essentially its metamodel (modulo its internal packaging structure) [1]. Then, the notion of model subtyping (i.e., safe replaceability) becomes easy to define [2, 3] and to check by tools [2].

However, the situation is not so bright for model transformations, mainly because of their dual nature: they can be considered to be both models and operations. Thus, the community must come up with new ideas and approaches for transformation typing. As models they can be naturally typed by the metamodel of their modeling language (e.g., QVT or ATL). However, typing them as operations is not easy. In general, specifying the type of any software artefact that exhibits behaviour (be it a function, operation, object, component, or a model transformation) is far from a trivial task, specially when its behaviour is rather complex. Furthermore, manipulating and reasoning about these kinds of behavioural types tend to be rather cumbersome and computationally expensive: normally these types try to capture the full behaviour of the artefact of interest independently from any context of use and require heavyweight reasoning techniques and tools, such as theorem provers.

Z. Hu and J. de Lara (Eds.): ICMT 2012, LNCS 7307, pp. 56–71, 2012.
© Springer-Verlag Berlin Heidelberg 2012

In this paper we discuss central ideas building the cornerstones of a light-weight and modular approach to model transformation typing, using *Tracts*. Tracts were introduced in [4] as a specification and black-box testing mechanism for model transformations. Thus every model transformation can be specified by means of a set of tracts, each one covering a particular scenario or *context of use*—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 partitioning the full input space of the transformation into smaller, more focused behavioural units, and to define specific tests for them. This approach to typing provides a form of "Duck typing" [5]. 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.

The organization of this paper is as follows. After this introduction, Section 2 presents model transformations, discussing the problem of model transformation typing, the issues of current approaches, and a frame which we use for formalizing model transformations. Then, Section 3 briefly presents Tracts, introducing its main characteristics and constituent elements. Section 4 describes our ideas underlying a light-weight approach to model transformation typing using sets of tracts, and discusses the kinds of analysis that are possible with our proposal, how to conduct them, as well as its current advantages and limitations. Finally, Section 5 compares our work to other related proposals and Section 6 draws the final conclusions and outlines some lines for future work.

2 Typing in MDE

2.1 Typing Models

Model types are useful in many MDE activities. For example, model types are needed for describing the signature (i.e., input and output parameters) of model operations and services, which in MDE are defined in terms of their metamodels. Thus, to perform an operation or a transformation on a model (conforming to a metamodel) we need to check first if it is a valid input for the operation. This situation is even more justified if modeling tools need to be connected, or for chaining several model transformations together. For connecting them, it is essential to check the type substitutability between the output of a service and the input of another, in such a way that type safety is guaranteed.

In our context, the type of a model is essentially its metamodel (modulo its internal packaging structure) [3]. Then we can consider that every metamodel M defines a collection of models $\mathcal{R}_M$ with the models that conform to M.

Let M and M' be metamodels (which can be considered as *types* for the sets of models that conform to them). We say that M' *extends* M (denoted by $M' <: M$) iff $\mathcal{R}_M \subseteq \mathcal{R}_{M'}$. In other words, $M' <: M$ implies that all models that conform to M also conform to M'. This is the equivalent operation to object subtyping in

object-oriented type systems [6], and therefore it also implies safe replaceability. This operator is also similar to the *matching* operator (<#) defined in [7, 3], although matching is in general weaker than subtyping.

For example, consider a metamodel `SM` to describe simple states machines, and another metamodel `Composite-SM` that adds to `SM` the possibility of allowing composite states [3]. Metamodel `Composite-SM` extends `SM` because every simple state machine can be considered as a particular case of composite state machine. Similarly, we could say that data type `Int` extends `Nat`, i.e., `Int` <: `Nat` because every value of `Nat` is a valid value of `Int`.

Intuitively, $M' <: M$ if: M' contains all classes and relationships in M; all attributes of M classes are present in the corresponding M' classes; and M' imposes the same or even stronger constraints to M elements than those that M imposes (including cardinality constraints). For a more complete definition we refer the reader to [2]. Note as well that the <: operator defines a *partial order*: It is Reflexive ($M <: M$), Transitive ($M' <: M \wedge M'' <: M' \Rightarrow M'' <: M$) and Asymmetric ($M' <: M \wedge M <: M' \Rightarrow M = M'$). It is partial because not any two metamodels can be related by this relation (e.g., the metamodels of simple state machines and of sequence charts).

2.2 Typing Model Transformations

Model transformation (MT) type systems are helpful in many situations. For example, they can be used to check that a transformation can be chained (or composed) with others, check that their behaviour is correct (w.r.t. its type), rule out transformations that would produce models that are not proper instance of their metamodels, identify useless transformations (e.g., that navigate never existing paths in a model), etc.

MT typing is not as easy to define as model typing because of the intrinsic "behavioural" aspects of model transformations, i.e., the way they transform model elements from the source metamodel into model elements of the target metamodel. We need to realize that model transformations comprise two different aspects: structure and behaviour. The former aspect defines the structural relation that should hold between source and target models, whilst the latter specifies how the specific source model elements are transformed into target model elements.

This intrinsic duality needs to be especially taken into account when reasoning about MT subtyping (or extensibility), which in our context has to do with *safe replaceability*. Replaceability refers to the ability of a software entity to substitute another, in such way that the change is transparent to external clients [8]. In the realm of model transformations, we say that transformation T' extends another model transformation T (and write $T' <: T$) if T' behaves as T with all valid input models of T. In other words, we will say that $T' <: T$ iff T' can safely replace T without being noticed by the clients of T.

We can identify at least three kinds of types for model transformations. In the first place we have the *language type*. The fact that model transformations are also models [9] provides one (naive) approximation to the problem of typing

transformations, by which the type of a MT is the metamodel of the language in which it is written (e.g, QVT or ATL). For example, the metamodel of QVT defines the set of all transformations written in that language.

Secondly, we have the *structural type*, defined by the fact that model transformations can be considered as operations, and therefore can be typed by the types of their input and output metamodels, as proposed in [10]. Then, if $T : M \to N$ is a transformation, its structural type is $M \to N$. With this, MT *structural subtyping* becomes similar to traditional subtyping of functions defined in type theory, which relies only on the contravariance of argument types and covariance of the return type.

More precisely, let $T : M \to N$ and $T' : M' \to N'$ be two transformations. We say that T' extends (or can structurally substitute) T ($T' <:_s T$) iff $(M' <: M) \wedge (N <: N')$. But again, this approach to typing model transformations is not sufficient [6]. It is like typing functions by their input and output parameters, or typing operations by their signatures. For instance, functions $\sqrt{x} : \texttt{Nat} \to \texttt{Nat}$ and $x^2 : \texttt{Nat} \to \texttt{Nat}$ become indistinguishable if we use this approach.

This is the reason for having to consider the *behavioural type* of a transformation T for defining it properly. In the most general case such a type needs to define how every valid input model is transformed into a valid output model. This can be specified in terms of a set $\mathfrak{S}$ of (*source*) constraints that defines the valid input models for T, a set $\mathfrak{T}$ of (*target*) constraints that defines the valid output models, and a set $\mathfrak{R}$ of (*source-target*) constraints that describe how individual source and target models should be related.

In other words, $\mathfrak{S}$ defines the preconditions that must hold for all input models of the transformation; $\mathfrak{T}$ defines the postconditions that must hold for the output models that the transformation produces; and $\mathfrak{R}$ defines conditions that should hold relating the individual source and target models. To express this, if $C[\![m]\!]$ means that a model m satisfies a constraint C (which is nothing but a logic predicate), then the *behavioural type* of a model transformation $T : M \to N$ is a triplet $(\mathfrak{S}, \mathfrak{T}, \mathfrak{R})$ such that: $\forall\, m \in \mathcal{R}_M \bullet \mathfrak{S}[\![m]\!] \Rightarrow \mathfrak{T}[\![T(m)]\!] \wedge \mathfrak{R}[\![(m, T(m))]\!]$.

To formally express behavioural subtyping, let $T : M \to N$ and $T' : M' \to N'$ two model transformations, for which $T' <:_s T$ (structural subtyping should be a requirement for behavioural subtyping), and let $(\mathfrak{S}, \mathfrak{T}, \mathfrak{R})$ and $(\mathfrak{S}', \mathfrak{T}', \mathfrak{R}')$ be the specification of the behavioural types of T and T', respectively. Then, T' can behaviourally substitute T ($T' <: T$) iff $(\mathfrak{S} \Rightarrow \mathfrak{S}') \wedge (\mathfrak{T}' \Rightarrow \mathfrak{T}) \wedge (\mathfrak{R}' \Rightarrow \mathfrak{R})$.

This is similar to Liskov's substitutability principle [6], which states that if S is a subtype of T ($S <: T$) , then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g., correctness). Liskov's principle imposes some standard requirements on signatures (adopted later in contract-based design [11]): contravariance of method arguments in the subtype; covariance of return types in the subtype; preconditions cannot be strengthened in a subtype; postconditions cannot be weakened in a subtype; and finally, invariants of the supertype must be preserved in a subtype.

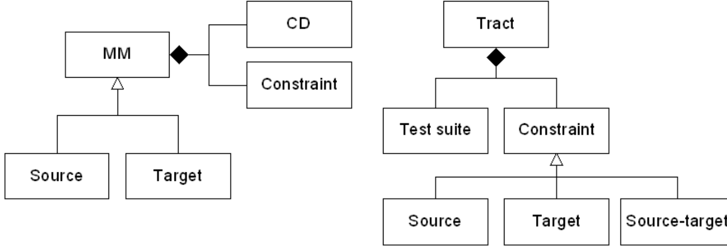

Fig. 1. Concepts in a Tract [4]

Although this definition of Model Transformation subtyping is appropriate from a theoretical point of view, it is not easy to check in practice without the aid of heavyweight tools such as theorem provers. Nevertheless, the given formulation admits one interesting way to check whether a model transformation T' can replace another one T, for which we know the behavioural type $(\mathfrak{S}, \mathfrak{T}, \mathfrak{R})$. It is enough to check that T' conforms to that type, i.e., $\forall\, m \in \mathcal{R}_M \bullet \mathfrak{S}[\![m]\!] \Rightarrow \mathfrak{T}[\![\, T'(m)]\!] \wedge \mathfrak{R}[\![(m,\, T'(m))]\!]$.

Using this notion of behavioural subtyping for model transformations we can search for a required transformation in a repository of transformations (such as `http://www.eclipse.org/m2m/atl/atlTransformations/`), or check that a given transformation can easily replace (or implement) another one, or that a given implementation of a model transformation conforms to its type (i.e., is correct w.r.t. its expected usage).

3 Tracts

One of the problems of the previous specification of MT behavioural type lies in its complexity. The specifications of an MT type can 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 excessive size. Because the type 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, checking the conformance of MT implementations and conducting other tests over these specifications become quite cumbersome, complex, and computationally prohibitive tasks.

In order to deal with the problems, we propose the use of *tracts*. They provide modular pieces of specification, each one focusing on a particular scenario. They have the structure of a behavioural type, plus a test suite that allows operationalizing the conformance tests. We do not provide the full behavioural 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

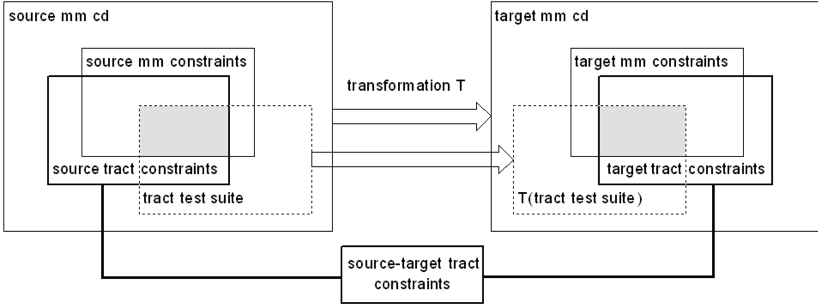

Fig. 2. Building Blocks of a Tract [4]

works in the rest of the cases. In this respect, this approach to typing is a form of "Duck typing": "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck" [5].

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. Such constraints serve as "contracts" (in the sense of contract-based design [11]) for the transformation in some particular scenarios, and are expressed by means of OCL invariants. Tracts are composed by conjunction, similarly to the modular specification of an operation using several pre- and postconditions, each one defining a specific situation or use case of the operation.

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 Fig. 2 we have displayed the central ingredients of our approach for transformation testing: a source and target metamodel, the transformation T under study, and a transformation contract, for short tract, which consists of a tract test suite and 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 contract, a so-called tract class. An example of a tract class called `2S1T-Tract` is shown later in this section.

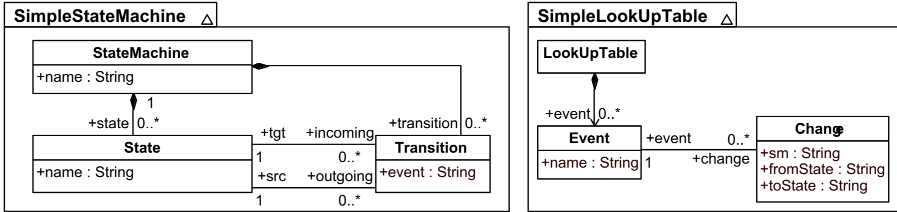

Fig. 3. Source and Target Metamodels of transformation `SM2T`

In Fig. 2, the rectangles indicate possible overlap (resp. disjointness) of source and target models. Basically, the tract — consisting of the test suite and the three kinds of constraints — checks for the correctness of the transformation in the sense that correct source models from the test suite are transformed into correct target models, i.e., our approach checks that in Fig. 2 the grey source section is transformed into the grey target section. In general, there can 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. We show the dashed rectangles with the test suites not necessarily inside source/target tract constraint rectangles in order to allow, e.g., the definition of negative tests for the transformation.

To test a transformation T against a tract t, the input test suite models can be automatically generated using languages like ASSL [12], and then transformed into their corresponding target models. These models can also be automatically checked with the USE tool [13] against the constraints defined for the transformation. The checking process can be automated, allowing the model transformation tester to process a large number of models in a mechanical way.

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

For illustration purposes, let us consider a model transformation `SM2T` between simple state machines and a lookup table that lists the events and their associated transitions [3]. The source and target metamodels of this transformation are shown in Figure 3. 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:

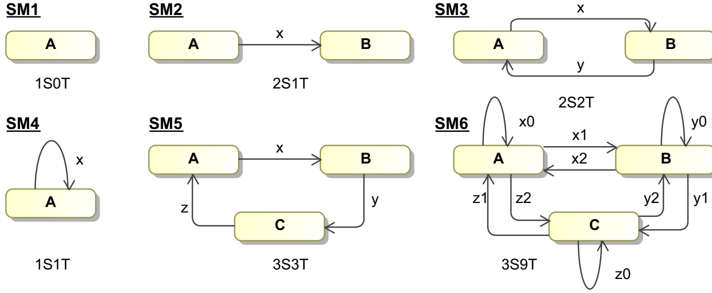

Fig. 4. Test suites samples for the 6 tracts defined for model transformation SM2T

```
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 4 (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.
- 2S2T: state machines with two states and two transition between them.
- 1S1T: state machines with single states and one transition.
- 3S3T: state machines with three states and three transitions, forming a cycle.
- 3S9T: state machines with three states and 9 transitions (see Figure 4).

Let us show here one of these 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:

```
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 (not shown here for space reasons).

4 Model Transformation Typing Using Tracts

Let us explain how (sub-)typing works for tracts. A tract is responsible for specifying how to transform a source model into a target model.

In Fig. 5 we see that `TractG` transforms metamodel `SourceG` into metamodel `TargetG`. '`G`' and '`S`' stand for 'general' (resp. 'specific'). `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.

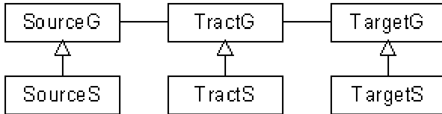

Fig. 5. Tract subtyping

Analogously this is the case for `TargetS`. `TractS` is a specialization of `TractG` and inherits from `TractG` its connecting associations. Constraints must guarantee that 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).

4.1 Tract Typing by Example

Fig. 6 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.

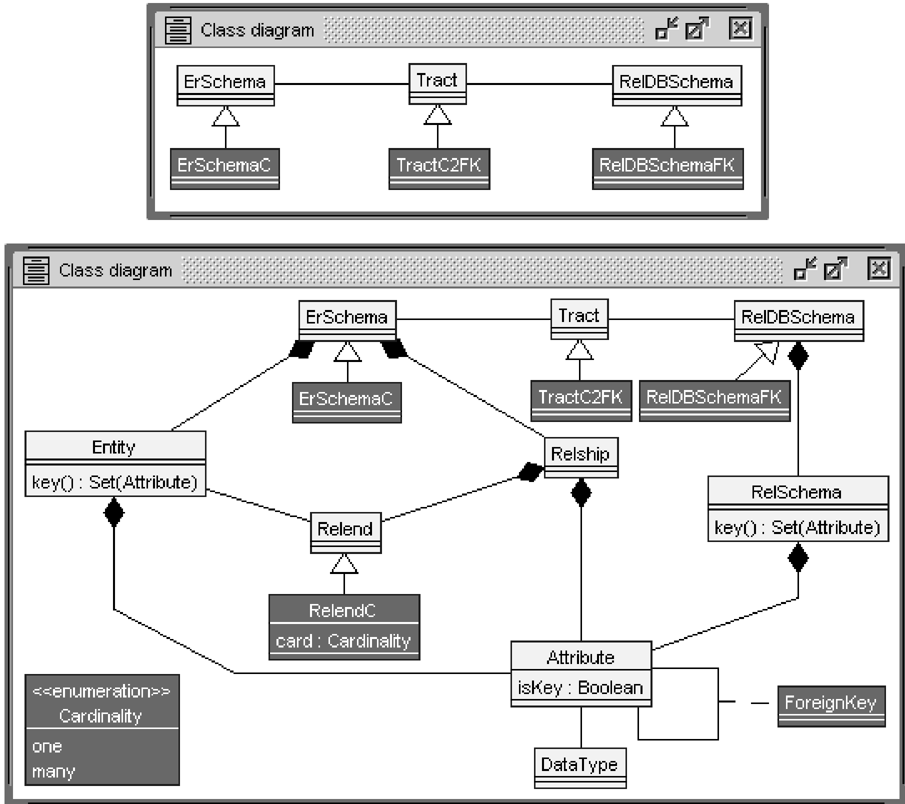

Fig. 6. An example of tract subtyping

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 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. 7 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

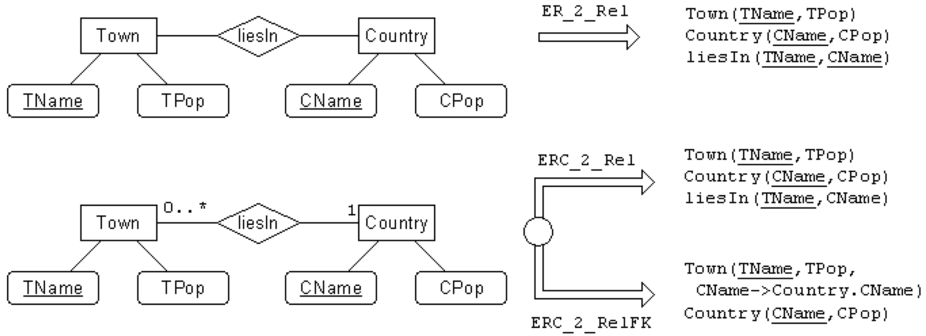

Fig. 7. `Town-liesIn-Country` scenario

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. 8 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`).

As shown in Fig. 9, 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` 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`.

4.2 Working with Tract Types

As mentioned above, the type of a model transformation T is specified in terms of a set of tracts $\{t_1, t_2, \ldots, t_n\}$. This section briefly discusses the kinds of analysis that can be conducted with tracts, as well as the pros and cons of our proposal.

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 [14] model transformation that implements the `SM2T` specification, and we need to test whether such MT is correct. This was the original intention of Tracts, and such a testing process is fully described in [4].

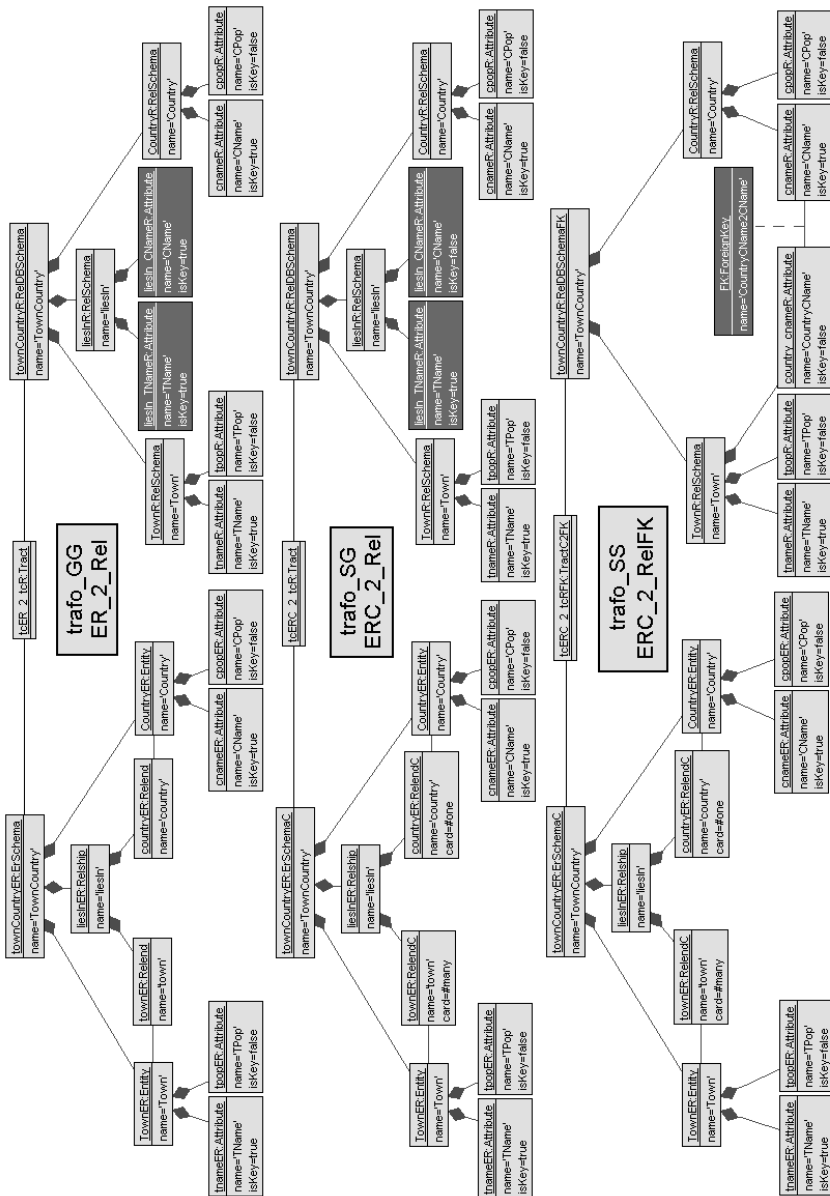

Fig. 8. `Town-liesIn-Country` object diagram

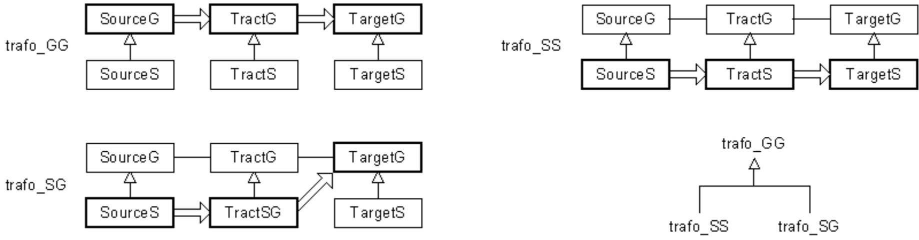

Fig. 9. Relationship between Example Transformations

Safe Substitutability of Model Transformations. Now, given another model transformation T', how do we decide if 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 [4]. This is a two step process: first of all a number of input models is automatically generated and then for each of these models we can check whether the transformation fulfills the associated tract. 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, and for the test suites of interest.

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: 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.

5 Related Work

There are several kinds of contributions that can be related to our work. In the first place we have the works that define contracts for model transformations, using different notations. One of the earlier works [15] introduces the concept of "transformation contract" in a similar way to ours—although without incorporating test suites. However, the authors propose to specify contracts by means of OCL operations, which causes many technical problems for writing contracts—as the own authors discuss in their paper. Besides, they do not discuss any practical way of using their contract specifications for model testing. The work in [16] also proposes OCL for defining transformation contracts, although 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. Finally, Poernomo [17] defines a similar proposal in spirit, but using constructive type theory instead of first-order logic.

Another proposal defines the type of a model transformation in terms of its input and output metamodels [10]. However, as mentioned in the introduction, such a *structural type* is not enough for capturing all relevant aspects: behaviour should also be taken into account. The work in [18] utilizes transformation types in an XML context.

The proposal presented in [19] is also of interest. The authors show how to derive some invariant-based verification properties that should be preserved by the transformation (which are similar to our tracts) by analysing the internal rules that compose a transformation. Although they follow a white-box approach to model transformation testing, it could probably be combined with ours if their approach could help us identify some more tracts for a transformation written in any of the languages they deal with (TGG and QVT). In this sense, we fully agree with one of the reviewer's suggestions about the interest to investigate the implications of the similarity between the tracts and these languages' transformation rules: when using tracts with these kinds of model transformations, is it a matter of typing using the existing rules, or is it necessary to have separate rules for implementation and specification, and if so, how structurally distinct should they be?

Furthermore, our idea of modularizing the specifications into smaller units could be transferred to other techniques apart from Tracts, e.g., to the invariant-based verification properties presented in [19] or to existing contract-based approaches [15, 16].

Another group of works (see, e.g., [20–24]) also use a white-box approach to model-transformation specification and testing, aiming at fully validating the behaviour of the transformation (including other properties such as confluence of the rules, termination, etc.) using formal methods and their associated toolkits—which include, e.g., Alloy, Maude, or graph rewriting techniques. Although more powerful than our approach from a theoretical perspective, their computational complexity generally makes them inappropriate for testing large model transformations. In addition, the drawback of a white-box approach is that it is tightly coupled to the transformation language and thus it would need to be adapted or completely redefined for another transformation language [25].

6 Conclusions

As MDE is becoming more widely applied and adopted, larger transformations are being developed, with thousands of lines of code. This makes them error-prone and brittle, becoming hard to understand, develop, debug, maintain and reuse. In fact, model transformations, like any other Software Engineering artefact, must be systematically designed and implemented [26]. The need to have effective mechanisms for specifying and properly testing them is now critical.

In this paper we have developed central ideas using *Tracts* for Model Transformation typing, and discussed benefits and limitations of this approach.

There are several lines of work that we plan to address next. For instance, we would like to study how to choose the tracts that compose the type of a model transformation, to ensure enough coverage and completeness. In this respect, we plan to investigate how to improve our proposal by incorporating some of the existing works on the effective generation of input test cases. We also plan to improve the current tool support for tracts, incorporating the creation and maintenance of libraries of tracts. Finally, larger case studies will be carried out in order to stress the applicability of our approach and to obtain more extensive feedback.

Acknowledgements. We would like to thank the anonymous reviewers for their constructive and insightful comments, and to Loli Burgueño and Fernando López for developing the tool support. This work is supported by Research Projects TIN2008-03107 and P07-TIC-03184.

References

1. Steel, J., Jézéquel, J.-M.: Model Typing for Improving Reuse in Model-Driven Engineering. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 84–96. Springer, Heidelberg (2005)
2. Romero, J.R., Rivera, J.E., Durán, F., Vallecillo, A.: Formal and tool support for model driven engineering with Maude. Journal of Object Technology 6(9), 187–207 (2007)
3. Steel, J., Jézéquel, J.M.: On model typing. Software and Systems Modeling 6(4), 401–413 (2007)
4. 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)
5. Heim, M.: Exploring Indiana Highways: Trip Trivia. Exploring America's Highway. Travel Organization Network (2007), `http://en.wikipedia.org/wiki/Duck_test`
6. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16(6), 1811–1841 (1994)
7. Bruce, K.B., Vanderwaart, J.: Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. ENTCS 20(1), 1–26 (2004)
8. Nierstrasz, O.: Regular types for active objects. In: Nierstrasz, O., Tsichritzis, D. (eds.) Object-Oriented Software Composition, pp. 99–121. Prentice-Hall (1995)

9. Bézivin, J.: On the unification power of models. Software and Systems Modeling (SoSyM) 4(2), 171–188 (2005)
10. Vignaga, A., Jouault, F., Bastarrica, M.C., Brunelière, H.: Typing in Model Management. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 197–212. Springer, Heidelberg (2009)
11. Meyer, B.: Applying design by contract. IEEE Computer 25(10), 40–51 (1992)
12. 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)
13. 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)
14. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
15. 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)
16. 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)
17. Poernomo, I.H.: Proofs-as-Model-Transformations. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 214–228. Springer, Heidelberg (2008)
18. Matsuda, K., Hu, Z., Takeichi, M.: Type-based specialization of xml transformations. In: Proc. of PEPM 2009, pp. 61–72. ACM (2009)
19. 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)
20. 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)
21. 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)
22. Küster, J.M.: Definition and validation of model transformations. Software and Systems Modeling 5(3), 233–259 (2006)
23. 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`
24. Troya, J., Vallecillo, A.: A rewriting logic semantics for ATL. Journal of Object Technology 10, 5:1–29 (2011)
25. 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)
26. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., Santos, O.: Engineering model transformations with transML. Software and Systems Modeling (2011) (in press)