Testing M2T/T2M Transformations

Manuel Wimmer[1] and Loli Burgueño[2]

[1] Business Informatics Group, Vienna University of Technology, Austria
wimmer@big.tuwien.ac.at
[2] GISUM/Atenea Research Group, Universidad de Málaga, Spain
loli@lcc.uma.es

Abstract. Testing model-to-model (M2M) transformations is becoming a prominent topic in the current Model-driven Engineering landscape. Current approaches for transformation testing, however, assume having explicit model representations for the input domain and for the output domain of the transformation. This excludes other important transformation kinds, such as model-to-text (M2T) and text-to-model (T2M) transformations, from being properly tested since adequate model representations are missing either for the input domain or for the output domain. The contribution of this paper to overcome this gap is extending Tracts [12], a M2M transformation testing approach, for M2T/T2M transformation testing. The main mechanism we employ for reusing Tracts is to represent text within a generic metamodel. By this, we transform the M2T/T2M transformation specification problems into equivalent M2M transformation specification problems. We demonstrate the applicability of the approach by two examples and present how the approach is implemented for the Eclipse Modeling Framework (EMF). Finally, we apply the approach to evaluate code generation capabilities of several existing UML tools.

1 Introduction

Much effort has been put into the establishment of model-to-model (M2M) transformation testing techniques in the past years [1,26]. Several approaches have been developed for defining contracts for M2M transformations that act as specifications for model transformation implementations [5,12], as oracle functions to validate the output of transformations [12,13], and as drivers for generating test cases [13]. In particular, constraints for input models, output models and for the relationship between both may be specified.

Besides M2M transformations, model-to-text (M2T) and text-to-model (T2M) transformations are of major importance in Model-driven Engineering [7]. M2T transformations are typically used to bridge the gap between modeling languages and programming languages by defining code generations but may be employed in a generic manner to produce text from models such as documentation or textual representations of a model's content. T2M transformations are typically used for reverse engineering [4], e.g., transforming legacy applications to models in the case of model-driven software modernization. However, these kinds of transformations have not gained much attention when it comes to testing.

In this paper we adopt current techniques for testing M2M transformations to the problem of testing T2M and M2T transformations. The prerequisite of using existing M2M transformation techniques is to have metamodels for the input and output of the transformations. However, for the side that is dealing with ”just” text, no metamodels are usually available. Even more problematic, when considering T2M and M2T transformations, a set of metamodels and T2M parsers may be required as a prerequisite. For instance, consider Web applications where in addition to a general purpose programming language several other languages may be employed where some of the languages are even embeddable in other languages. Thus, developing metamodels and T2M parser support for such complex settings may introduce a huge overhead.

To alleviate the burden from T2M and M2T transformation developers, we introduce a generic approach that may be used for any transformation task where text is involved as input or output of the transformations. The main mechanism we employ is to represent text within a generic metamodel in order to transform M2T and T2M transformation specification problems into equivalent M2M transformation specification problems. The proposal is combinable with any contract-based M2M transformation approach, but in this paper we demonstrate its application with Tracts [12].

The structure of the paper is as follows. The next section introduces Tracts, a M2M transformation testing approach, by example. Section 3 shows how to represent text-based artifacts as models to allow for reusing the M2M transformation testing approaches. Section 4 demonstrates how Tracts are defined for M2T and T2M transformations and gives details about the implementation of the approach. Section 5 presents an evaluation of the approach, in particular to explore its capabilities to find shortcomings in code generations delivered by current UML tools. In Section 6 we present related work and in Section 7 we conclude the paper with an outlook on future work.

2 Tracts for Testing Model-to-Model Transformations

Let us shortly introduce the formalism used in this paper, namely Tracts, for specifying M2M transformation contracts. As we shall see, these formalism assumes to have metamodels for the input and for the output of the transformation as all other existing contract specification approaches do.

Tracts were introduced in [12] as a specification and black-box testing mechanism for model transformations. They provide modular pieces of specification, each one focusing on a particular transformation scenario. 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 partitioning 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 Tracts is that we do not require complete proofs, just to check that the transformation works for the tract test suites, hence providing a *light-weight* form of verification.

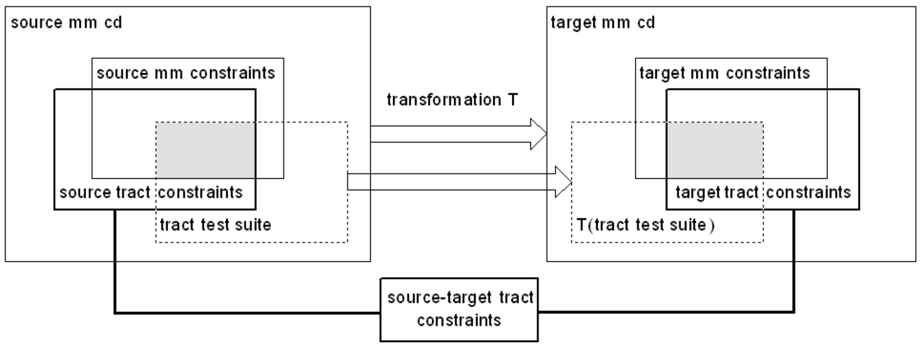


Fig. 1: Building blocks of a tract.

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 [19]) for the transformation in some particular scenarios, and are expressed by means of OCL invariants. They provide the *specification* of the transformation.

In Figure 1 more details about the tracts approach are presented. The necessary components the approach rely on are the source and target metamodels, the transformation *T* under test and the transformation contract, which consists of a tract test suite and a set of tract constraints. In total, five different kinds of constraints are present: the source and target metamodels are restricted by general constraints added to the language definition, and the tract imposes additional source, target, and source-target tract constraints for a given transformation.

If we assume a source model *M* being an element of the test suite and satisfying the source metamodel and the source tract constraints given, the tract essentially requires that the result *T(M)* of applying transformation *T* satisfies the target metamodel and the target tract constraints and the tuple *<M, T(M)>* satisfies the source-target tract constraints.

For demonstrating how to use Tracts, we introduce the simple transformation example *Families2Persons*.[3] The source and target metamodels of this transformation are shown in Figure 2. For this example, a set of tracts is developed to consider only those families with exactly four members (mother, father, daughter, son):

```
-- C1: SRC_oneDaughterOneSon
 Family.allInstances->forAll(f|f.daughters->size=1 and f.sons->size=1)
-- C2: SRC_TRG_Mother2Female
 Family.allInstances->forAll(fam|Female.allInstances->exists(f|
   fam.mother.firstName.concat('␣').concat(fam.lastName)=f.fullName))
-- C3: SRC_TRG_Daughter2Female
 Family.allInstances->forAll(fam|Female.allInstances->exists(f|
   fam.daughters->exists(d|d.firstName.concat('␣').concat(fam.lastName)
     =f.fullName)))
```

[3] The complete example is available at our project website `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Tracts`

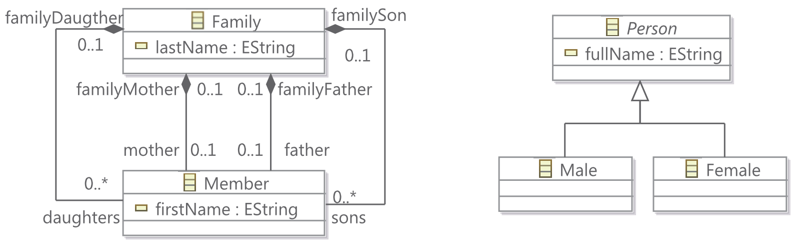


Fig. 2: The Family and Person metamodels.

```
-- C4: SRC_TRG_MemberSize_EQ_PersonSize
 Member.allInstances->size=Person.allInstances->size
-- C5: TRG_PersonHasName
 Person.allInstances->forAll(p|p.fullName<>'' and
    not p.fullName.oclIsUndefined())
```

Concerning the kinds of the shown Tracts, C_1 represents a pre-condition for the transformation, $C_2 - C_4$ define constraints on the relationships between the source and target models, i.e., constraints that should be ensured by the transformation, and finally, C_5 represents a post-condition for the transformation. Note that this approach is independent from the model transformation language and platform finally used to implement and execute the transformation.

3 A Generic Metamodel for Text

In order to reuse M2M transformation specification and testing approaches, we have to transform the M2T or T2M transformation specification problem into a M2M transformation specification problem. For this, the text artifacts residing in the input or output domain of the transformations under study have to be injected to the model engineering technical space [16].

For realizing this goal, there are several options. We may either decide to go for a specific format conforming to a specific grammar or to use a generic format that is able to represent any text-based artefact. In case there is already a metamodel available for the specific grammar, then this metamodel may be a good choice anyway. However, for most transformation scenarios involving text at one side there are no metamodels available, because metamodels are often not required at all. Just consider the case of generating documentation from models. Although there is no generalized and fixed structure, it may be necessary to check certain requirements of the transformation. This is why we have decided to use the second option, which allows us to save upfront the effort required when developing M2T or T2M transformations in general. Furthermore, using a generic metamodel to represent the text artifacts also reflects best practices in the development of M2T transformations, where no metamodel is used for the text artifacts. For example, consider template-based M2T transformation languages[4]. Usually,

[4] http://www.omg.org/spec/MOFM2T/

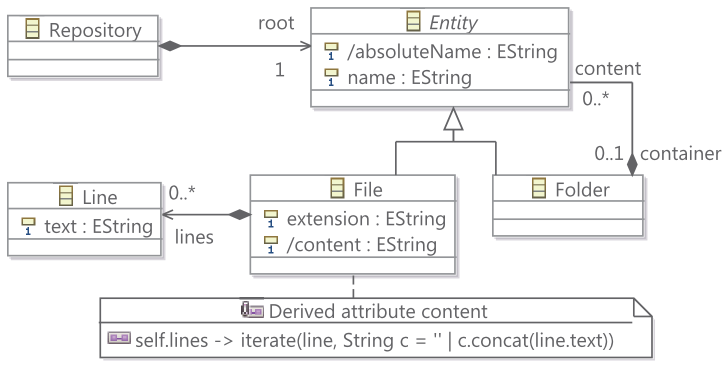


Fig. 3: Metamodel for representing text artifacts and repositories.

template-based approaches are used to generate that text. Finally, even if there is a T2M parser, this is again a transformation that may have to be specified and tested. Thus, our generic approach may be used to test the specific approach.

Apart from this, there is a second aspect that needs to be considered when dealing with text-based artifacts. The artifacts are normally organized in a hierarchical folder structure, which should be taken into account. For instance, the output of a M2T transformation may not be just a single file but several, which should also be arranged in a certain folder structure. Thus, our approach has to cover concepts for describing the structure of a repository that contains the input or output artifacts of a transformation.

Figure 3 shows the metamodel for representing text artifacts stored in repositories using a certain structure. Meta-class Repository represents the entry point to the root folder containing folders and files or to a file if only one single artefact is used. While folders just contain a name, files have in addition an extension as well as a content. The content of files is represented by lines that are sequentially ordered. A derived attribute content is used to allow easy access to the complete content of a file.

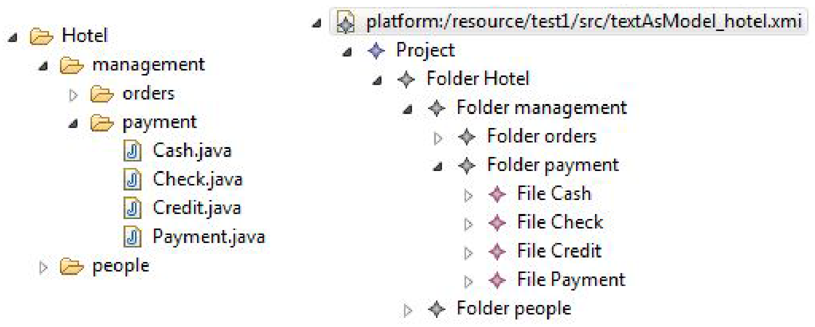
Figures 4 and 5 present an instance of the text metamodel coming from a Java code repository. On the left hand side of the figures the Java folder structure as well as the content of a Java file are shown, while on the right hand side an excerpt of the corresponding text model (shown in the EMF tree browser) is illustrated.

4 M2T/T2M Transformation Testing By-Example

This section shows how the metamodel for describing text artifacts can be used in conjunction with tracts for M2T and T2M transformation testing.

4.1 M2T Example: UML to Java

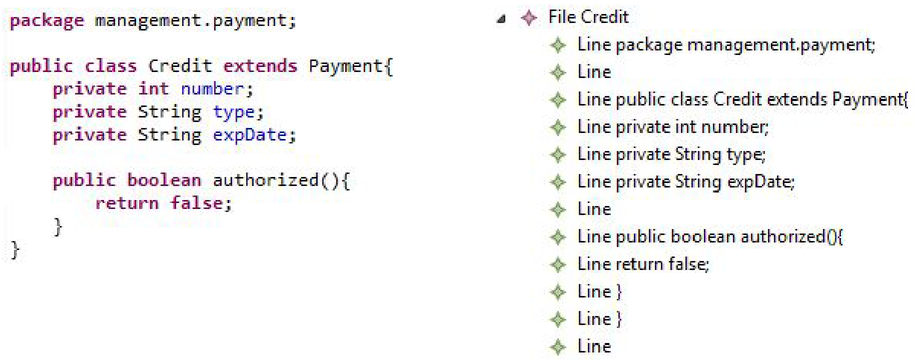
For illustration purposes, let's apply our approach to a given case: the transformation that converts UML class models into the corresponding Java classes—which are text



```
Hotel
  management
    orders
    payment
      Cash.java
      Check.java
      Credit.java
      Payment.java
  people
```

```
platform:/resource/test1/src/textAsModel_hotel.xmi
  Project
    Folder Hotel
      Folder management
        Folder orders
        Folder payment
          File Cash
          File Check
          File Credit
          File Payment
      Folder people
```

Fig. 4: Exemplary folder structure and corresponding text model.



```java
package management.payment;

public class Credit extends Payment{
    private int number;
    private String type;
    private String expDate;

    public boolean authorized(){
        return false;
    }
}
```

```
File Credit
  Line package management.payment;
  Line
  Line public class Credit extends Payment{
  Line private int number;
  Line private String type;
  Line private String expDate;
  Line
  Line public boolean authorized(){
  Line return false;
  Line }
  Line }
  Line
```

Fig. 5: Exemplary file content and corresponding text model.

files that should be stored in folders inside a code repository. Figure 6 shows the subset of the UML metamodel that we will consider in this scenario. The target metamodel is the one that we described above for speficying text artifacts, and that was shown in Figure 3.

The specification of such a transformation is composed of a set of tracts, each one focusing on a particular property that we want to ensure. As illustrative examples we have chosen 10 tracts, which are described below. Notice that in some of them we have used auxiliary operations such as toFirstUpper and toString to clarify the code. How these auxiliary operations are defined as an user-defined library in OCL is explained in Subsection 4.3.

The first tract states that nested UML packages should be transformed into nested folders. This is specified by the following constraint:

```
-- C1: Nested packages are transformed into nested folders
 Package.allInstances() -> forAll(p| Folder.allInstances()->
   exists(f| f.name = p.name and p.subPackages->
      forAll(subp | f.folders()->exists(subf | subf.name = subp.name))))
```

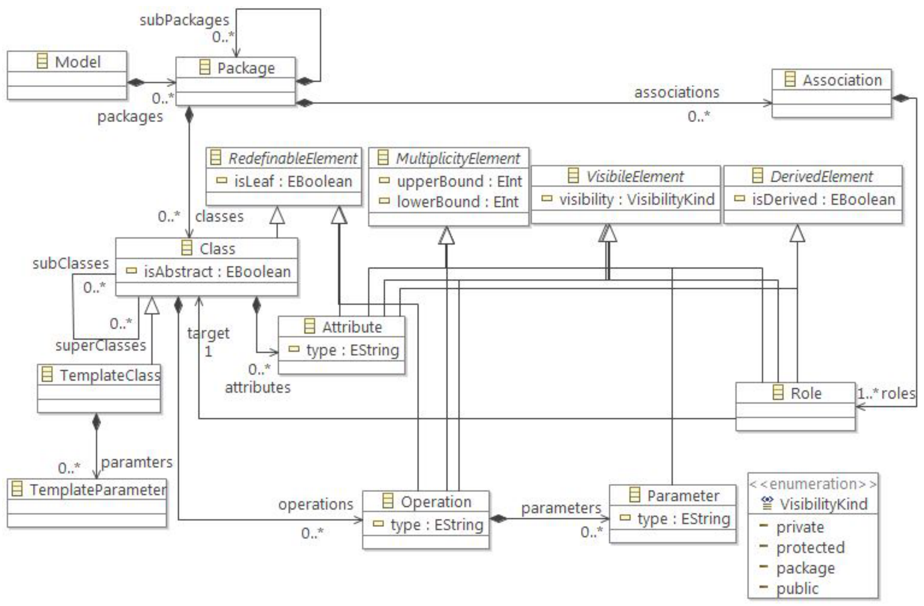


Fig. 6: A simplified metamodel for UML class diagrams.

The second tract states that Java packages should be imported when associations occur between elements contained in different UML packages.

```
-- C2: Import of packages when associations are crossing package borders
  Association.allInstances -> select(a |
  a.roles->at(1).target.package <> a.roles->at(2).target.package )
  -> forAll(a| File.allInstances->exists(f |
    f.name = a.roles->at(1).target.name and f.extension = 'java' and
    f.content().matchesRE('import.*'+ a.roles->at(2).target.name))))
```

We should also ask for a precondition in order not to allow that any class inherits from a leaf class.

```
-- C3: No leaf class as superclass
Class.allInstances() -> forAll(c| c.isLeaf implies c.subClasses->isEmpty())
```

Another precondition should check that there is no multiple inheritance in use in the UML model (multiple inheritance is not allowed in Java).

```
-- C4: Only one superclass allowed in Java
  Class.allInstances()->forAll(c | c.superClasses->size()<=1)
```

We also include here some tracts to specify how particular elements in the UML model should be transformed. For example, derived attributes can not be modified in Java, and therefore only getter methods are generated for them.

```
-- C5: Derived attributes only have a getter method
Class.allInstances->forAll(c| File.allInstances
  ->exists(f | f.name = c.name and f.extension = 'java' and
    c.attributes->select(a | a.isDerived)->forAll(a |
    not f.content().matchesRE(a.type+'.*?'+a.name+'.*?;') and
    f.content().matchesRE(a.type+'\\s+get'+ toFirstUpper(a.name)))))
```

Similarly to the tract above, the following tract specifies how the visibility of attributes should be handled by the transformation.

```
-- C6: Visibility of attributes is considered
Package.allInstances->forAll( p|
  p.classes->forAll(c | File.allInstances->exists(f |
    f.name = c.name and f.extension = 'java' and
    f.container.name = p.name and
    c.attributes->forAll(a |
        f.content().matchesRE(toString(a.visibility)
            +'.*?'+a.type+'.*?'+a.name+'.*?;')))))
```

And the same for association ends:

```
-- C7: Visibility of roles is considered
 Association.allInstances->forAll(a | File.allInstances->exists(f |
 f.name = a.roles->at(1).target.name and f.extension = 'java' and
  f.content().matchesRE(toString(a.roles->at(2).visibility)+
    '.*?'+a.roles->at(2).target.name+'.*?'+a.roles->at(2).name+'.*?'))))
```

Finally, three further constraints specify that there are no Java keywords in the UML models, that the names of the elements and folders are well formed (e.g., no control characters), and that generic UML classes are supported.

```
-- C8: No keywords as name in UML model
NamedElement.allInstances()->forAll(ne | not Set{'abstract',
  'extends','implements','class','public','private','protected',...}
  .includes(ne.name))
-- C9: Well-formed names
NamedElement.allInstances()->forAll(ne |
    ne.name.matchesRE('[a-zA-Z_][a-zA-Z0-9_]*'))
-- C10: Generic classes are supported
TemplateClass.allInstances->forAll(c | File.allInstances->exists(f |
    f.name=c.name and f.extension='java' and
    f.content().matchesRE('class\\s+'+c.name+'\\s+<.*?>.*?{'))))
```

Of course, further constraints can be defined to deal with other requirements on the transformation. We have included here the tracts above in order to show examples of the expressiveness of our approach in the case of an M2T transformation. We do not try to claim completeness of full coverage of our specifications for the UML to Java case.

4.2 T2M example: USE to UML

To illustrate the applicability and usage of our proposal in the case of T2M transformations, we have focused on a transformation between textual USE [11] specifications of structural models, and its corresponding UML specifications. USE features for representing models are similar to the ones defined in UML: classes, attributes, associations and operations. For example, the following USE code corresponds to a simple model of persons owning cars.

```
 class Person
   attributes
       name : String
       birthDate: Integer
   operations
       age() : Integer
 end
 abstract class Vehicle
   attributes
       brand : String
 end
```

```
class Car < Vehicle
   attributes
       licenceNumber : String
end
association Person_Car between
   Person [0..1] role owns
   Car [*] role owner
end
```

The following set of constraints are examples to show how different requirements on the transformation from USE to UML can be stated.

The first constraint specifies that the USE model should reside in only one file.

```
-- D1: Only one file per transformation run allowed
 File.allInstances()->size() = 1
```

The second constraint states that every USE class will correspond to one UML class with the same name.

```
-- D2: Every USE class should result in UML class
  Line.allInstances()->select(l | l.text.matchesRE('^\\s*class'))->
    forAll(l|Class.allInstances->exists(c|l.text.matchesRE(c.name)))
```

The third one specifies how USE inheritance relationships (cf. '<' symbol in the USE example) are transformed into UML inheritance relationships.

```
-- D3: less-than sign has to open an inheritance relationship
Line.allInstances()->select(l | l.text.matchesRE('\\s*class.*<'))->
 forAll(l|Class.allInstances->exists(c | l.text.matchesRE(c.name) and
 c.superClasses->exists(superClass|l.text.matchesRE(superClass.name))))
```

Similarly, the last three constraints allow to specify that USE abstract classes are transformed into UML abstract classes, USE attributes into UML attributes, and USE associations into UML associations.

```
-- D4: USE abstract classes to UML abstract classes
Line.allInstances()->select(l|l.text.matchesRE('abstract\\s+class'))->
  forAll(l|Class.allInstances->
  exists(c|l.text.matchesRE(c.name) and c.isAbstract))
-- D5: USE attributes to UML attributes
Class.allInstances()->forAll(c|c.attributes->
  forAll(a|File.allInstances->asSequence()->first().content().
  matchesRE('class\\s*'+c.name+'\\s*(<\\s*[A-Za-z0-9]+)?\\s*attributes.*?'
  +a.name+'\\s*:\\s*'+a.type+'.*?(end|operations)')))
-- D6: USE associations to UML associations
Association.allInstances->forAll(a |
File.allInstances->asSequence()->first().content().matchesRE(
  a.roles->iterate(r; s : String =
        '(association|composition)\\s+[A-Za-z0-9_]+\\s+between.*?' |
        s.concat(r.target.name+'.*?role␣'+r.name+'.*?'))))
```

4.3 Tool Support

In order to provide tool support for our proposal, we have developed a *injector* (parser) that converts the content of a repository, i.e., files, folders, and their structure, into a model that conforms the Text metamodel shown in Figure 3, and an *extractor* that takes models conforming to the Text metamodel and produces text organized in folders.

In order to check that a given M2T transformation fulfils a set of constraints (such as the ones shown in Section 4.1), we run the transformation with the set of models defined by the tract *test suite* (these input models have not been shown before for the sake of simplicity) and then use the injector with the output text (organized in folders) resulting from the transformation to generate the corresponding output models conforming to the Text metamodel. Then we are in a position to check the validity of the constraints as in the case of tracts defined for M2M transformations, with our TractsTool [26]. The TractsTool evaluates the defined constraints on the source and target models by a transparent translation to the USE tool [11].

The case of testing T2M transformations is similar. Here the test suite is defined by the tract as a set of repositories, which need to be transformed first into a model-based representation by our injector component to check the source constraints. When the source constraints are fulfilled, the content of the repository is transformed by the T2M transformation under test to produce the output models. The models produced from the repository and their corresponding output models can then be validated by the TractsTool against the tracts.

For easing the formulation of the OCL constraints, we have also enriched USE with a set of libraries and operations to deal with Strings. For instance, to deal with regular expressions in OCL we have introduced the matchesRE() operation shown above that checks whether a given sequence matches a regular expression or not. Furthermore, we have also introduced some auxiliary functions that are currently provided by M2T transformation languages such as toFirstUpper() to end up with more concise OCL constraints than just using the standard OCL String operation library.

The TractsTool for testing M2T/T2M transformations is available at our project website[5] with several examples.

5 Evaluation

Most UML tools provide code generation facilities to produce source code from UML models. In order to evaluate the usefulness of using contract-based specifications of code generators, we tested a selected set of currently available UML tools by checking a set of tracts.

5.1 Selected Tracts and Test Models

For the evaluation, we used the constraints defined by the tracts presented in Section 4.1, which represent some of the most essential requirements that any UML to Java code generator has to fulfil. These constraints are described below, together with their type ('Scr' for source constaints, 'Trg' for target constaints and 'ScrTrg' for source-target constaints), as well as one example of the test suite models that were used to check the tracts.

- C_1 SrcTrg: Nested packages are transformed into nested folders. Minimal test model: two nested packages in a UML model.
- C_2 SrcTrg: Import of packages supported. Minimal test model: two packages, each one having one class, both connected by an association.

[5] `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Tracts`

C_3 Src: Inheritance of a leaf class is not allowed. Minimal test model: a class inheriting from a leaf class.

C_4 Src: Only single inheritance is used in UML. Minimal test model: one class having two superclasses.

C_5 SrcTrg: Derived attributes only result in getter method. Minimal test model: one class having one derived attribute.

C_6 SrcTrg: Visibility of attributes mapped to Java. Minimal test model: one class having one public, one private, one package, and one protected attribute.

C_7 SrcTrg: Visibility of roles mapped to Java. Minimal test model: two classes related by three associations, whose association ends have different visibilities (public, private, package, and protected).

C_8 Src: No Java keywords are allowed as names in UML models. Minimal test model: one class with name "class", one attribute with name "public", and one operation with name "implements".

C_9 Src: Names in UML model have to be valid Java identifiers. Minimal test model: one class with name "-", attribute with name "+", and operation with name "?".

C_{10} SrcTrg: Generic classes mapped to Java. Minimal test model: one generic class with two parameters.

5.2 Selected Tools

We selected six UML tools from industry that claimed to support code generation from UML class diagrams into Java code. The selected sample covers both commercial tools and open-source projects.

- **ArgoUML** (`http://argouml.tigris.org`) is a modeling tool supporting UML 1.4 diagrams. It is an open source project and distributed under the Eclipse Public License (EPL). Currently there is only one edition of ArgoUML available. We evaluated version 0.34.
- **Poseidon for UML** (`http://www.gentleware.com`) is a modeling tool supporting UML 2.0, distributed by Gentleware. We evaluated the community edition of Poseidon for UML, version 6.0.2.
- **MagicDraw** (`http://www.nomagic.com`) is a commercial modeling tool supporting UML 2.0 and is distributed by NoMagic. We evaluated the enterprise edition, version 16.8.
- **EnterpriseArchitect** (`http://www.sparxsystems.com`) is a commercial modeling tool supporting UML 2.4.1 and is distributed by SparxSystems. We evaluated the professional edition, version 10.
- **BOUML** (`http://www.bouml.fr/`) is a UML 2.0 diagram designer which also allows for code generation. We evaluated version 4.22.2.
- **Altova UModel** (`http://www.altova.com/umodel.html`) is a UML 2.0 tool for software modeling. We evaluated Altova UModel 2013, the latest version available.

Table 1: Evaluation results

TOOL	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
ArgoUML	✓	✓	×	×	-	✓	✓	×	×	✓
Poseidon	✓	×	×	✓	×	✓	✓	×	✓	-
MagicDraw	✓	✓	✓	✓	×	✓	✓	×	✓	✓
EnterpriseArchitect	✓	✓	✓	×	×	✓	✓	×	×	✓
BOUML	×	✓	-	✓	×	✓	✓	×	✓	✓
Altova UModel	×	✓	✓	✓	×	✓	✓	✓	✓	✓

5.3 Evaluation procedure

We defined reference test models based on UML metamodel shown in Figure 6. Subsequently, we re-modelled the reference test models in all of the selected tools. Having the models within the specific tools allowed us to run the validation support and code generators of the specific tools. The validation support is related to the evaluation of support for the *Src* constraints that should act as filter for the code generator, i.e., only valid models should be transformed to code. Thus, we validated all test models in case validation support is available in a specific tool and checked if validation errors or at least warnings are reported for the negative test models associated to the *Src* constraints. For checking *Trg* and *SrcTrg* constraints, we translated the output of the code generators to Text models and evaluated the resulting output in combination with the input models, i.e., the reference models, using the testing approach described in this paper.

It has to be mentioned that the UML tools are delivered with standard configurations for the code generators. Some tools also allow to tweak the code generation capabilities by configuring certain options using specific wizards before running the code generation. Others also allow to edit the code generation scripts, enabling further possibilities to customize the code generation facilities beyond the possibilities offered by the wizards. In this sense, we evaluated first the standard code generation features the tools offer, and after that we tried to tweak the tools by using the wizards to fulfill additional constraints that were not fulfilled in the standard configuration. However, the customization possibilities based on the wizards could not enhance further the evaluation results for the given constraints.

5.4 Results

Table 1 shows the results of the evaluation. In the table, a tick symbol (✓) means that the test passed for that tract and a cross symbol (×) means that the tract test failed. Some of the tests were not available for a given tool, e.g., a particular modeling feature is missing, and were not performed. This is indicated by a dash (-).

In the first place, constraint C_1 did not hold for some tools. In the case of BOUML and Altova UModel, the code generation requires that UML elements are manually associated to certain artifacts for which a path must be specified. Thus, the user has to specify the folders and Java files that should be generated. All other tools work well with packages in an automated way.

Concerning associations that cross package borders (C_2), Poseidon is the only tool that does not take this feature into account.

Precondition C_3 checks that no class inherits by another class marked as leaf. BOUML does not include the option to set a class as leaf. Poseidon fails because it lets that a class inherits from a leaf class. ArgoUML passes the test and give a warning during the model validation only when the superclass is marked as leaf before the creation of the generalization relationship.

C_4 checks that the UML model does not use multiple inheritance, because it cannot be used in Java. ArgoUML and EnterpriseArchitect fail because they do not check this constraint, and they both create a Java class which does not even compile.

Concerning C_5, ArgoUML does not allow to define derived features. The rest of the tools do, but derived features are ignored in the code generation process. An expected solution would create derived attributes into their corresponding getter methods.

All tools work well with the transformation of the visibility of attributes and roles (constraints C_6 and C_7).

Most tools fail with constraints C_8 and C_9 (use of Java keywords and invalid names in Java). Tools do not seem to conduct any validation check before the code generation starts. Although many tools allow several kinds of validation checks on the UML models, most of these tests only deal with UML constraints. A few tools also allow the development of user-defined validation checks, but they do not seem to have been defined for the code generation facilities they support. The only exception is Altova UModel, which raises a warning if non-valid Java identifiers are used as names for UML elements.

Finally, generic classes are supported and correct Java code is generated by all UML tools (constraint C_{10}) except Poseidon, which does not allow to define generic classes.

In summary, the results show that code generators have to fulfill several properties that should be specified at a higher level for allowing their validation. In particular, we found that no tool performs well even with respect to the basic UML to Java code generators. Furthermore, we discovered that several tools produced incorrect Java code, even not compilable in some situations. In this sense, the tracts presenting the basic requirements could be used as the initial components of a benchmark for future improvements and developments of UML-to-Java code generators.

6 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 [2,3,8,23]. 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 M2M transformations [1,26]. However, as mentioned before, transformations involving text on one side have not been extensively studied.

Several kinds of works apply contracts for M2M transformation testing using different notations for defining the contracts. In the following, we divide them into 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.

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 [9,15,17,18]. Model comparison frameworks are employed for computing a difference model between the expected and the actual target models. If there are differences then it is assumed that there exists an error either in the transformation or in the source/target model pair. Analogously, one could employ text comparison frameworks to reason about an expected text artefact and an computed text artefact. However, reasoning about the cause for the mismatch between the expected and actual text artefact solely based on the difference model is challenging. Several elements in the difference model may be effected by the same error, however, the transformation engineer has the burden to cluster the differences by herself.

A special form of verification by contract was presented in [20]. The authors propose to use model fragments (introduced in [22]) which are expected to be included in a target model which is produced from a specific source 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. Using our text metamodel, one is able to define such fragments even for M2T/T2M transformations, but they still only define the oracle for one particular input model.

In previous work [13] we proposed a declarative language for the specification of visual contracts 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. This approach may be used as an alternative syntax for our presented approach. Further alternative text-based approaches for defining oracles are presented in [5,6,9,10,14], however, they do not discuss how to apply their approaches for text artefacts.

The most closely related work is presented in Tiso et al. [25] where the problem of testing model-to-code transformations is explicitly mentioned. The authors enumerate two possibilities for such tests. First, they briefly mention a static approach which evaluates if certain properties are fulfilled by the transformation target code. However, they do not describe the details of this possibility. Second, they discuss a dynamic approach based on checking the execution of the transformation target, which is subsequently elaborated in their paper. In particular, they model, in addition to the domain classes, test classes that execute certain operations and check for given post-conditions after the operations have been executed. While we propose a generic and static approach to test M2T/T2M transformations in general, Tiso et al. propose an approach for testing a specific model-to-code transformation, namely from UML class diagrams to specific Java code and using JUnit tests that are also derived from a model representation. Furthermore, in our approach we have the possibility to directly test M2T/T2M transformations. However, in Tiso et al. [25] the execution output of the generated application has to be analyzed to trace eventual errors back to the M2T transformation.

Finally, an approach for testing code generators for executable languages is presented in [24]. The authors present a two-folded approach. On the one hand, first-order test cases that represents the models which are transformed into code are distinguished. On the other hand, second-order test cases are introduced representing tests that are executed on models as well as on the derived implementation, i.e., on the generated code. The output of the code execution is compared with the output of the model execution. If these outputs are equivalent, it is assumed that the code generators works as expected. Compared our proposal, we provide an orthogonal approach for testing the syntactic equivalence by checking certain constraints, i.e., how to define oracles for the first-order test cases. Combining a syntactical with a semantical approach seems to be an interesting subject for future work.

7 Conclusions and Future Work

This paper presented a language-agnostic approach for testing M2T/T2M transformations. Agnostic means independent from the languages used for the source and target artifacts of the transformations, as well as to the transformation language used for implementing the transformations. By extending OCL with additional String operations, we have been able to specify contracts for practical examples and evaluated the correctness of current UML-to-Java code generators offered by well-known UML tools. This evaluation showed a great potential for further improving code generators and documents the real need for an engineering discipline to develop M2T/T2M transformations.

There are several lines of work that we would like to explore next. In the first place, we plan to investigate how current Architecture Driven Modernization (ADM)[6] modeling standard such as Knowledge Discovery Metamodel (KDM) [21] may be used for defining contracts that are programming language independent and reusable for a family of code generators. For example, the presented contracts may be platform independently expressed and reused for testing UML-to-C# code generators. Secondly, the TractsTool we have used is a prototype whose limits need to be explored and improved. The models defined in the Tracts' test suites are normally of reasonable size (less than one or two thousand elements) because this is usually enough for checking the Tract constraints. However, we have discovered that large models (with several thousands of model elements) are hard to manage with the tools that we currently use. In this sense, looking for internal optimizations of the tool is something we also plan to explore next. Finally, we are working on the development of a benchmark for UML-to-Java code generators that could be useful to the community, based on a modular approach such as Tracts and on the proposal presented in this paper.

[6] `http://adm.omg.org`

Acknowledgements. This work is partially funded by Research Project TIN2011-23795. We would like to sincerely thank the *Bremen Database Systems Group* led by Prof. Martin Gogolla, in particular to Lars Hamann, for their excellent support and help with their tool USE.

References

1. 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: Proceedings of the 1st International Workshop on Verification and Validation of Model Transformations (VOLT 2012) @ ICST. pp. 921–928. IEEE (2012)
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: Proceedings of International Workshop on Integration of Model Driven Development and Model Driven Testing (IMDD-MDT 2006) @ ECMDA (2006)
3. Baudry, B., Ghosh, S., Fleurey, F., France, R., Traon, Y.L., Mottu, J.M.: Barriers to Systematic Model Transformation Testing. Commun. ACM 53(6), 139–143 (2010)
4. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: a generic and extensible framework for model driven reverse engineering. In: Proceedings of the 25th International Conference on Automated Software Engineering (ASE 2010). pp. 173–174. ACM (2010)
5. Cariou, E., Belloir, N., Barbier, F., Djemam, N.: OCL contracts for the verification of model transformations. ECEASST 24 (2009)
6. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: Proceedings of the International Workshop on OCL and Model Driven Engineering @ MODELS (2004)
7. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–646 (2006)
8. France, R.B., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: Proceedings of the 29th International Conference on Software Engineering (ISCE 2007) - Future of Software Engineering Track. pp. 37–54. IEEE Computer Society (2007)
9. 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: Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS 2011). LNCS, vol. 6981, pp. 395–409. Springer (2011)
10. Giner, P., Pelechano, V.: Test-Driven Development of Model Transformations. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009). LNCS, vol. 5795, pp. 748–752. Springer (2009)
11. 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)
12. Gogolla, M., Vallecillo, A.: *Tract*able Model Transformation Testing. In: Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA 2011). LNCS, vol. 6698, pp. 221–235. Springer (2011)
13. Guerra, E.: Specification-driven test generation for model transformations. In: Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT 2012. LNCS, vol. 7307, pp. 40–55. Springer (2012)
14. Kolovos, D., Paige, R., Rose, L., Polack, F.: Unit testing model management operations. In: Workshop Proceedings of the IEEE International Conference on Software Testing Verification and Validation (ICSTW 2008). pp. 97–104. IEEE Computer Society (2008)

15. Kolovos, D.S., Paige, R.F., Polack, F.A.: Model comparison: a foundation for model composition and model transformation testing. In: Proceedings of the International Workshop on Global Integrated Model Management (GaMMa 2006) @ ICSE. pp. 13–20. ACM (2006)
16. Kurtev, I., Bézivin, J., Akşit, M.: Technological spaces: An initial appraisal. In: Proceedings of the Confederated International Conferences (CoopIS, DOA, and ODBASE), Industrial track (2002)
17. Lin, Y., Zhang, J., Gray, J.: Model comparison: A key challenge for transformation testing and version control in model driven software development. In: Proceedings of the Workshop on Best Practices for Model-Driven Software Development @ OOPSLA. pp. 219–236 (2004)
18. Lin, Y., Zhang, J., Gray, J.: A testing framework for model transformations. In: Beydeda, S., Book, M., , Gruhn, V. (eds.) Model-Driven Software Development – Research and Practice in Software Engineering. pp. 219–236. Springer (2005)
19. Meyer, B.: Applying design by contract. IEEE Computer 25(10), 40–51 (1992)
20. Mottu, J.M., Baudry, B., Traon, Y.L.: Model transformation testing: oracle issue. In: Workshop Proceedings of the IEEE International Conference on Software Testing Verification and Validation (ICSTW 2008). pp. 105–112. IEEE Computer Society (2008)
21. Pérez-Castillo, R., de Guzmán, I.G.R., Piattini, M.: Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. Computer Standards & Interfaces 33(6), 519–532 (2011)
22. Ramos, R., Barais, O., Jézéquel, J.M.: Matching Model-Snippets. In: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007). LNCS, vol. 4735, pp. 121–135. Springer (2007)
23. Straeten, R.V.D., Mens, T., Baelen, S.V.: Challenges in Model-Driven Software Engineering. In: Models in Software Engineering. LNCS, vol. 5421, pp. 35–47. Springer (2008)
24. Stürmer, I., Conrad, M., Dörr, H., Pepper, P.: Systematic testing of model-based code generators. IEEE Trans. Software Eng. 33(9), 622–634 (2007)
25. Tiso, A., Reggio, G., Leotta, M.: Early Experiences on Model Transformation Testing. In: Proceedings of the 1st Workshop on the Analysis of Model Transformations (AMT 2012) @ MODELS. pp. 15–20. ACM (2012)
26. Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal Specification and Testing of Model Transformations. In: Advanced Lectures of the 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems - Formal Methods for Model-Driven Engineering (SFM 2012). LNCS, vol. 7320, pp. 399–437. Springer (2012)