JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Towards interactive, test-driven development of model transformations

Jesús Sánchez Cuadrado
Universidad de Murcia, Spain

ABSTRACT Developing model transformations is a complex task because it requires a deep knowledge of the semantics of the input and output meta-models. Many times this knowledge is refined as the transformation is developed. A simple approach to encode this knowledge is in the form of test cases, consisting of pairs of input models and expected output models. However, creating these test cases is a time consuming and error-prone activity and it is barely used. Moreover, there is little tool support for refining a transformation interactively at the same time that the test suite is created.

This paper explores an approach for test-driven development of model transformations based on creating test cases using a model finder in an interactive manner. The synthesis of the input models of the test cases is driven by the results of static analysis with respect to the coverage of the transformation, with the goal of instantiating input models which are not yet handled by the transformation. The expected models are automatically derived using an instrumented transformation, and the user is in charge of inspecting the result to validate them. The approach has been implemented for ATL and integrated into AnATLyzer using USE Model Validator as the backend model finder.

KEYWORDS Model transformation, Test-driven development, Model finding, USE Model Validator.

1. Introduction

Model transformations (MT) are an essential part of Model-Driven Engineering since they allow developers to manipulate models automatically, for instance to achieve tool interoperability, to convert models to different formalism for analysis, to create refactorings or to animate models. Model transformations can be developed using model transformation languages (MTL), like ATL (Jouault et al. 2008), ETL (Kolovos et al. 2008), QVT (OMG 2005), Henshin (Arendt et al. 2010), etc. The design of a model transformation language typically includes constructs specifically tailored to implement transformations in an efficient way. At the same time, a model transformation language can be accompanied with a development environment (IDE) which provides productivity features such as editors with syntax highlighting, static analysis (Cuadrado et al. 2016) and quick fixes (Cuadrado et al. 2018b). Moreover, there are also approaches to automate and facilitate the creation and execution of tests cases from transformation specifications (Gogolla & Vallecillo 2011)(Guerra & Soeken 2015). However, despite of these facilities, the development of model transformations is still a complex and error-prone activity.

Using current technology model transformations are developed in a top-down fashion and with little testing (Baudry et al. 2010)(Guerra et al. 2019). There exists methods to design transformations (Guerra, de Lara, Kolovos, et al. 2013) but they are seldom used. In practice, in the author's experience the source to target mappings and the required transformation rules are "discovered" incrementally by building examples of concrete models (even if these models are constructed informally). At the same time, building input test models manually is a tedious and error prone activity, and therefore it is typically not carried out. Alternatives like creating transformation contracts and generating test cases from them can also be cumbersome, since they require a good knowledge of the semantics input and output metamodels.

This paper proposes a method to develop model transforma-

JOT reference format:
Jesús Sánchez Cuadrado. *Towards interactive, test-driven development of model transformations.* Journal of Object Technology. Vol. 19, No. 3, 2020.

http://dx.doi.org/10.5381/jot.2020.19.3.a18

tions incrementally and interactively, growing the set of input test models semi-automatically as the transformation is developed. In contrast to other testing approaches (González & Cabot 2012; Fleurey et al. 2004) we aim at generating input models which are not covered yet by the transformation. The goal is that they serve as guidance to evolve the transformation by inspecting concrete examples. The method is supported by a tool for ATL, integrated into ANATLYZER[1]. Technically, we combine static analysis and model finding, in particular model synthesis and partial model completion, to suggest relevant input models and help the developer understand the corner cases of the transformation at the same time that a proper test suite is constructed. To achieve this we make use of the USE Model Validator (Kuhlmann & Gogolla 2012; Gogolla et al. 2018) (USE MV), a robust model validator, which is integrated in Eclipse through EFINDER[2] (Cuadrado & Gogolla 2020).

Altogether, this paper makes the following contributions:

- An approach to incrementally develop a model transformation as well as its test suite by automatically synthesizing uncovered models.
- Tool support for ATL integrated into ANATLYZER which includes: static analys, input model synthesis, test suite execution and maintenance, and inspection of test cases.
- This work can also been regarded as a case study about how to use model finding, in particular USE MV features, to enhance model transformation tooling.

Organization. This paper is organized as follows. Section 2 introduces a running example and motivates this work. Section 3 presents the proposed development process. Section. 4 and Sect. 5 explains the details about our static analysis and the generation of input models respectively, whereas Sect. 6 explains how test cases are inspected and maintained. Section 7 shows how the approach is implemented in practice. Finally, Sect. 8 presents the related work and Sect. 9 concludes.

2. Background

The availability of techniques to ensure the correctness of model transformations is essential for the success of MDE. The fact that model transformation languages are higher-level that general purpose language could, in principle, provide opportunities for supporting verification methods which are more difficult to provide in general-purpose language languages. For instance, several transformation-specific analysis have been proposed for different languages (Cuadrado et al. 2018a)(Cheng & Tisi 2017)(Oakes et al. 2018), mechanisms for generating test cases automatically (González & Cabot 2012)(Guerra & Soeken 2015)(Gogolla & Vallecillo 2011), visualizations (Guana & Stroulia 2014), etc.

An scenario which has been less explored until now is to enhance the development environment with features to help the transformation developer construct meaningful transformation rules interactively by means of (semi-)automatically generated test cases. Instead of generating a complete test suite directly,

[1] http://anatlyzer.github.io
[2] https://github.com/jesusc/efinder

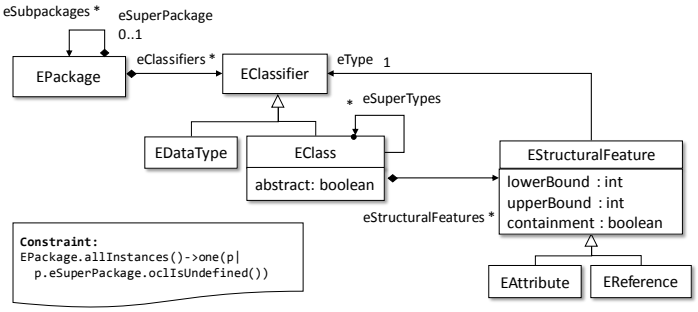

Figure 1 Excerpt of the Ecore meta-model.

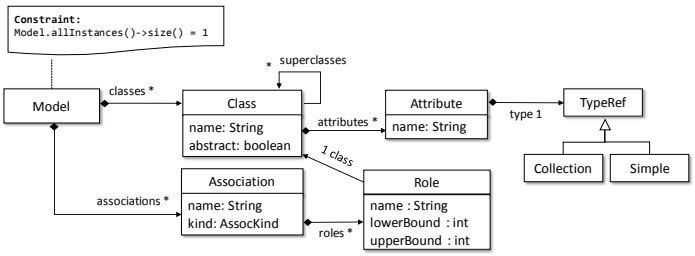

Figure 2 Excerpt of the USE meta-model.

in this proposal the test suite is grown incrementally as the transformation is developed, thus allowing the developer to reason over concrete example models.

2.1. Running example

As a running example let us consider an ATL transformation to convert Ecore meta-models into USE class diagrams[3]. Fig. 1 shows an excerpt of the Ecore meta-model and Fig. 2 shows a meta-model for USE.

The Ecore meta-model includes a constraint to express that a meta-model contains a unique root package[4]. The USE meta-model requires that there is only one Model element, to reflect the fact that in USE each specification contains a single class diagram.

2.2. Motivation

Developing a model(-to-model) transformation is typically a complex task because the developer must know the semantics of both the source and target domains of the transformation (in the example, the details of Ecore and USE class diagrams) and how they related to each other. Moreover, the developer must ensure that the implementation handles all possible input configurations correctly. This is done by creating transformation rules with the appropriate input patterns or by documenting invalid configurations with pre-conditions. At the same time, the target models generated by the transformation must be both syntactically and semantically correct by creating adequate output patterns and relationships between rules. This includes ensuring that the transformation satisfies the invariants of the meta-models as

[3] Full transformation and meta-models available at http://atenea.lcc.uma.es/projects/MTB.html
[4] This is technically not requirement for an Ecore meta-model to be valid, but it is a common assumption.

well as transformation post-conditions (i.e., constraints over the target meta-models specific to the models generated by the transformation).

A well known approach to establish the semantics of a transformation is to create transformation contracts (Tracts (Gogolla & Vallecillo 2011)(Hilken et al. 2018) is a specific instance of this approach), which specifies the main relationships between the input and target meta-models. For instance, the following listing is a piece of contract for the running example, taken from the Ecore2USE case study in (Burgueño et al. 2014). It specifies that there must be the same number of classes in the source and target models and it uses the class name as a way to identify them.

```
1 -- Same number of classes
2 EClass.allInstances→size = Class.allInstances→size
3
4 -- Equivalent names
5 EClass.allInstances→forAll(ec |
6   Class.allInstances→exists(uc | uc.name = ec.name))
```

Listing 1 Contract for Ecore2USE.

From these contracts, test cases can be generated automatically and used to create a test harness. As noted in (Gogolla & Vallecillo 2011), the contracts do not need to be exhaustive (i.e., cover all configurations), but it can be refined gradually. Nevertheless, this approach forces the developer to have a "top-down" knowledge of the transformation, in the sense that he or she needs to be able to describe the mapping in an abstract way. Actually, the proposed contract does not respect the Ecore semantics since it assumes that the class name is not scoped by its package name. This is a corner case, which is typically difficult to consider upfront without seeing a concrete example. Moreover, the original contract did not include constraints to indicate how Ecore sub-packages are handled (i.e., in USE nested models/packages are not allowed). Although contracts are certainly useful to reason about transformation semantics, it can be difficult to identify corner cases, which may require reasoning about concrete instances.

A simpler alternative is to create pairs of input/output test cases manually which, in the process of creating them, may help the developer identify and showcase the corner cases. This can be considered a "bottom-up" approach. The main advantage is that it allows us to work with concrete examples about the expected behaviour of the transformation, which may help in the implementation task. Unfortunately, creating these pairs of models can be very time consuming and this approach is seldom used.

In this paper we propose a development process that allows a transformation developer to create transformation in a bottom-up manner by growing a test suite incrementally and interactively, as new transformation rules are created. This approach is not incompatible with existing approaches, like the use of transformation contracts. On the contrary, it can also be used to refine existing contracts and gain a better knowledge of the transformation semantics iteratively.

2.3. A quick tour

This section provides a quick tour of our tool for the test-driven development of ATL transformations. The goal is to give the reader a concrete view of our proposal in order to facilitate the understanding of the following sections.

Fig. 3 shows how a developer would start developing the running example. The different steps are illustrated with screenshots from our tool. First, an initial rule is created (label 1), from which an initial test case is derived (label 2). The developer inspects the the test case (label 3), which is a passing test case because the output is as expected, and marks it as reviewed. Then, the transformation is analysed to find out which elements are not covered by the transformation (label 4). In this case, EClass is not covered and it is choosen by the user to generate a new test case (label 5). This implies synthesizing an input model and executing the transformation to generate the output model (label 6). The developer inspects the test case and find outs that the output model should contain two objects of type USE Class (label 7). This means that the transformation is not behaving as expected. If we fix the transformation (labels 8 and 9) and re-run it, the output model is as expected and the developer commits the output model to the test suite by promoting it to "expected model" (label 10).

In summary, the purpose is to avoid the burden of manually creating test cases. The input models are automatically created, and can be refined manually if needed. The output models are generated automatically by running the transformation. The developer commits the output model when it is correct, and there is no need to manually create it.

Next section gives an overview of the proposed method and the rest of the sections describes its technical realization.

3. Interactive, test-driven development transformations

Our proposal for interactive, test-driven model transformations is based on three ingredients: the use of static analysis including a coverage analysis, model finding to enhance static analysis and to automatically extend the test suite with new test cases, and finally, automated execution of the test suite so that user can inspect the output of new test cases and identify transformation refinement needs.

In the case of the running example, to start developing the transformation from scratch, we could begin by writing the simplest rule that we could come up with. For instance, we could write a rule to map Ecore packages to USE models, and set simple properties like name and classes. The following listing shows the initial version of the transformation.

```
1 rule package2model {
2   from e : Ecore!EPackage
3   to c : Use!Model (
4     name <- e.name,
5     classes <- e.eClassifiers
6   )
7 }
```

Listing 2 Ecore2USE transformation (version #1)

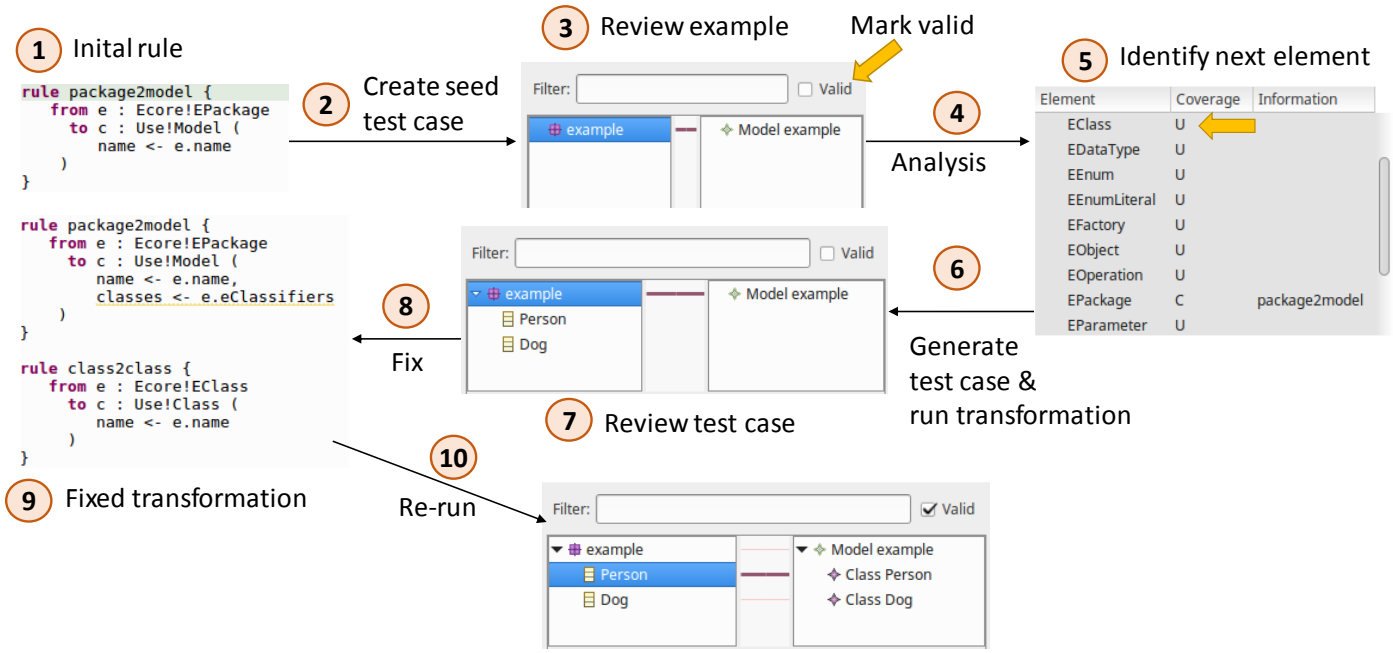

Figure 3 Usage of the interactive transformation tool.

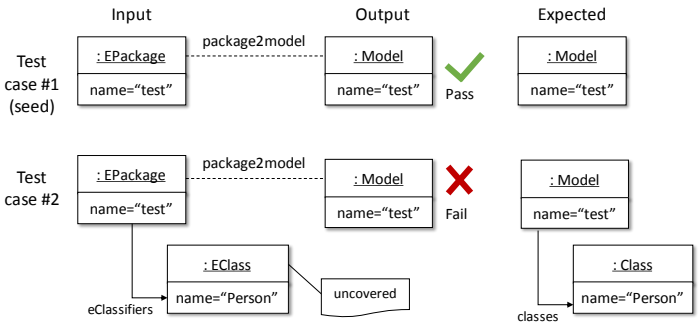

Figure 4 Test suite examples. Input models (left), expected models (right) and transformation output (middle).

We can start testing the transformation by writing a simple test case. Figure 4 shows the seed test case of our test suite. From this we woud like to know which input configurations have not been handled yet and to create new test cases which include these elements. For instance, in Listing 2, there are not any rule to handle EClass elements. This information can be extracted by analysing the coverage of the transformation rules with respect to the meta-model elements.

In an ATL transformation we can find three scenarios of "lack of coverage". (1) there might be rules yet to be written (missing rules), (2) existing rules might have missing bindings (e.g., target features not yet assigned), and (3) it might be that some of the bindings are not fully covered by existing rules. To analyse the coverage of the transformation rules with respect to the input meta-model, we have implemented a new analysis in ANATLYZER, called *Uncovered elements*. It checks which meta-model elements have not been handled by the transformation. For instance, elements like EClass, EClass.abstract and EAttribute have not been handled yet by the transformation. This is described in more details in the next section. In addition to this analysis, it is possible to use existing analysis provided by ANATLYZER in order to generate example input models, namely:

- *Unresolved bindings*. Checks whether a binding is completely covered by the transformation rules. If a binding is not resolved, the element right-hand side of the binding will not be resolved into an element assignable to the target feature. For instance, the binding in line 5 will be unresolved because there is no rule with EClassifier as input pattern (or two rules for covering EClass and EDataType which are the direct subtypes of EClassifier).
- *Invariant analysis*. It checks whether any possible output model of a transformation will satisfy the post-conditions, target meta-model invariants and transformation contracts expressed in OCL (Cuadrado et al. 2017). In the running example, the invariant in the USE meta-model will be violated by the transformation. The analysis generates a counter example which can be used as a test case as well.
- *Unconnected components*. Checks if all the target elements generated by the transformation form a connected graph. If this is not the case, then it might be sign of missing bindings or rules. For instance, with the current version of the transformation, a model with a root EPackage and a children EPackage element will in result an unconnected graph (i.e., two unlinked Model elements). This model will also be discovered by the invariant analysis in this case because the USE meta-model has a precise constraint about this.

The goal is then to devise a development process driven by the results of these analysis. The process would help the developer construct the transformation by incrementally covering more input meta-model elements and by examining new pairs of

input/output models (i.e., a generated test case). We propose the development process depicted in Fig. 5. For simplicity, the process assumes that the transformation is going to be developed from scratch and using a deterministic transformation language. It has the following steps:

1. A new transformation is created. To begin with, a simple rule is created. Typically, to transform the root element. In the running example we have started with package2model.

2. Create one or more initial test cases. This step can be performed manually or automatically by generating a seed input model which covers all the elements of the transformation, and then execute automatically the transformation to generate the corresponding output model. The initial test case is a seed test case to start growing the test suite. The test suite is formed by pairs of input and output models. Given that the initial rule is very simple, it is assumed to be correct and the output model is therefore a valid expected output model.

3. We use ANATLYZER to analyse the transformation and identify a number of elements that are not covered by the current set of rules by using the coverage analysis mentioned above. For instance, elements like EClass, EPackage.eSubPackages are not covered in the current version of the transformation.

4. The user inspects the results of the analysis and choose a meta-model element to cover in the next iteration. For instance, in the example a natural selection would be EClass.

5. The system synthesizes one or more input models which include the choosen meta-model element. The synthesis is constrained to produce models that do not include the paths for which the transformation already cover the element. If we choose EClass the resulting input model could be similar to the test case #2 of Fig. 4.

6. The transformation is executed automatically to show the user both the input and outputs of the transformation. The input model can be refined manually if needed. The user inspects the output model to determine if this is what it is expected. If not, then it means that the transformation needs to be fixed. If the output model is considered a valid expected model (w.r.t. to the input model), then the input/output pair is added to the test suite, and the process continues. In the example, test case #2 is not yet valid because its output model is not as expected. We need to add a rule for EClass to generate the expected output model.

7. The test suite is run continously as part of the process, to ensure that fixing a specific test case does not break existing ones (or alternatively, to fix existing test cases). Running the test suite consists of executing the transformation using the input model of each test case, and comparing the resulting output model with the expected model of the test case.

8. The process finishes when the input meta-model is completely covered, or when the developer determines that the uncovered elements are not of interest for the transformation.

A key element is to provide comprehensive tool support for the process. The developer should be able to perform all manual steps with the less possible effort. In particular, the inspection of the generated test case must be easy and fast. To this end, test cases tend to be small and focused on the new elements to be tested. In addition, the model transformation is instrumented to generate a trace model (as a secondary target model) which includes information about which meta-model elements has been "touched" by the transformation. This information allows us to display runtime coverage information from the test cases and the explicit mapping between the input and output models.

The next section presents the analysis that we perform and Sect. 5 describes how to use model finding to generate new test cases.

4. Transformation analysis

In the proposed process, we make use of several analysis which have been briefly discussed above. In this section we discuss the *uncovered elements analysis*, which is particularly useful for the interactive process. First, we discuss the analysis for classes and then we extend it for structural features.

4.1. Uncovered classes

This analysis is in charge of checking which classes of the input meta-models are not handled by any rule, or if there are some instances of a class which are handled but others not. We say that a class is covered when, for each valid input model, each instance of the class is matched by at least one input pattern[5].

Our approach to identify which classes are not covered by a transformation is based on two steps. In the first step we identify classes that are fully covered or partially covered, and in a second step we use model finding to reason about the rule filters.

Gathering initial coverage information. Algorithm 1 shows the first step, which classifies input metamodel classes into fully and partially covered. A class is *fully covered* when there is an input pattern whose type is the same class or a super type, and there is no condition involved in its matching. A class is *partially covered* when there is also a matching input pattern, but it includes conditions (e.g., in rule filters). In addition, abstract classes requires special treatment since they are covered only when all its concrete subclasses are also covered (lines 14 – 19).

Listing 3 shows an evolution of the running example. In this transformation class EClass is fully covered by rule class2class, whereas class EPackage is partially covered by rule package2model. To determine whether EPackage is not actually fully covered (e.g., if there are pre-conditions which complements the rule filter to cover the whole input configuration) we need to resort to model finding to reason about the OCL expressions.

[5] We say "at least" instead of "exactly one" because we also consider lazy rules.

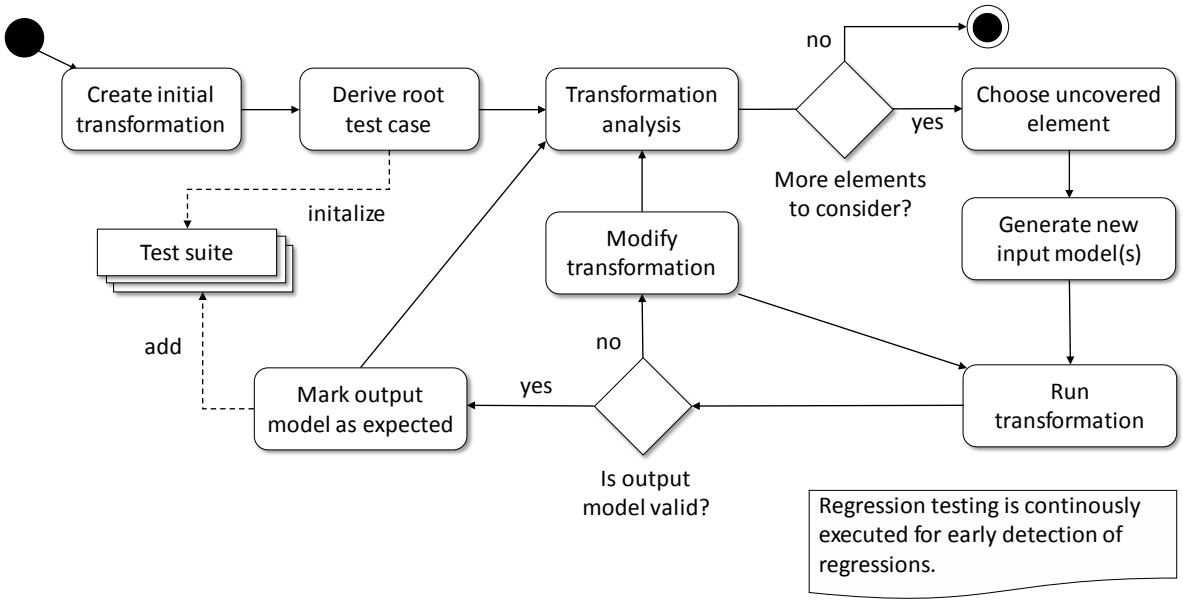

Figure 5 Activities in the proposed development process.

```
rule package2model {
  from e : Ecore!EPackage (e.eSuperPackage.oclIsUndefined())
  to c : Use!Model (
    name <− e.name,
    classes <− e.eClassifiers
  )
}

rule class2class {
  from e : Ecore!EClass
  to c : Use!EClass (
    name <− e.name
  )
}
```

Listing 3 Ecore2USE transformation (version #2).

Reasoning about OCL expressions. We use model finding to determine if a class, initiallly deemed as partially covered, is actually fully covered. To this end, we need to construct a formula which aggregates all possible path conditions for partially covering the class (in the example there is only one path), and we check if the union of the paths is *not equal* to the full set of objects of the class. A path condition is an expression which describes the features of the input models which will make the transformation execution reach the desired execution point. The details about how to construct path conditions in ATL are detailed in (Cuadrado et al. 2016). In this case, if there is a model that satisfies the path condition, it means there is at least one execution path that is not considered by the transformation. Moreover, we need to consider the transformation pre-conditions and the input meta-model invariants. For instance, to determine if EPackage is covered, we feed USE MV with the following set of constraints (Listing 4). In this case, USE MV answers with a counter-example, indicating that EPackage is not fully covered (i.e., a package with nested packages is not covered).

```
−− Invariant
EPackage.allInstances()→one(p |
  p.eSuperPackage.oclIsUndefined())

−− Coverage criteria
EPackage.allInstances()→select(p |
  p.eSuperPackage.oclIsUndefined()) <> EPackage.allInstances()
```

Listing 4 Constraints to determine the coverage of EPackage in the running example.

In practice, if an object of a given type is not matched by any input pattern, it means that such object is not transformed. There are three scenarios:

- It can be a smell that the transformation is incomplete, and our system helps the developer identify these cases and add new logic to handle them.
- It can be the case that such input configuration is not intended to be handled by the transformation, and thus must be forbidden. Writing a pre-condition is an adequate way to document this situation. In this example, we could restrict the transformation to work with metamodels with a single package using this pre-condition: EPackage.allInstances()->size() = 1. In this case, EPackage would be fully covered.
- An alternative situation is that the developer wants to deliberately ignore these objects, because they do not belong to scope of the transformation but they are valid input elements. For instance, a transformation that only transform UML Activity Diagrams. In ATL there is not a way to create rules that "ignore" elements. In other languages, notably RubyTL (Cuadrado et al. 2006), it is possible to write "ignore rules" to make this knowlege explicit.

4.2. Uncovered structural features

The analysis for structural features that is proposed in this work is simple and just relies on the previous analysis. If a class is *uncovered* all its structural features are uncovered as well. If a class is *fully or partially covered*, we need to detect the usages of the structural features of the class within the rule. If a feature

Algorithm 1 Identification of uncovered elements.

Input: Input meta-model classes
Output: Set of pairs (class, rules) with partial coverage
Output: Set of pairs (class, rules) with full coverage

```
concreteClasses ← {c ∈ classes | not c.isAbstract}
 foreach r in rules do
   inClass ← r.inPattern
    foreach c in concreteClasses do
      if c = inClass or c is a subtype of inClass then
        if r has filter then
          add (c, r) to partial coverage
        else
          add (c, r) to full coverage
            break
   end
end
abstractClasses ← {c ∈ classes | c.isAbstract}
 foreach c in abstractClasses do
   if if all concrete subclasses of c are in full coverage then
     rules ← rules associated to subclasses add (c, rules) to
       full coverage
   else if if all concrete subclasses of c are in full or partial
     coverage then
     rules ← rules associated to subclasses add (c, rules) to
       partial coverage
end
```

is used as part of the rule we consider it covered, otherwise uncovered. This analysis can be more precise by considering path conditions as above, but this is left for future work

5. Creating new test cases

Given the analysis described in the previous sections, the next step is to use the information about which elements have not been covered yet as a means to construct new test cases. Our goal is to both incrementally evolve the transformation and to improve the test suite. The rationale is as follows: if the developer is provided with a new test case, then he or she has a concrete instance to reason about, and this will help in the task of creating new rules or refine existing ones in order to evolve the transformation in the direction of increasing the coverage.

5.1. Generating input models

In this step, the user is presented with the list of uncovered classes. He or she has to select one of the classes to create a new test case which covers it. Our basic approach is to create new input models which contain at least one object not covered by the transformation.

For fully uncovered classes this is relatively straightforward. For instance, to consider EAttribute, we just need to use EAttribute.allInstances()->size() > 1.

For partially uncovered classes, we need to generate path conditions in which the filter conditions are negated. For instance, for EPackage we would generate the following constraint.

```
EPackage.allInstances()→exists(p | not p.eSuperPackage.
    oclIsUndefined())
```

Listing 5 Constraint to generate a new input model with nested EPackage elements.

To create models which are structurally more meaningful we need to add constraints to enfore the containment hierarchy given by the meta-model. This means, that an object must only exists within its container if its class is not a root class. In this case, we would add to the previous constraint:

```
EClass.allInstances()→forAll(c |
 EPackage.allInstances()→exists(p | p.eClassifiers→includes(c)))
and
EDataType.allInstances()→forAll(c |
 EPackage.allInstances()→exists(p | p.eClassifiers→includes(c)))
and
EStructuralFeature.allInstances()→forAll(c |
 EClass.allInstances()→exists(p |
   p.eStructuralFeatures→includes(c)));
```

Listing 6 Constraints to consider the containment hierarchy.

Finally, to consider structural features and to give more diversity to the generated models, we could extend the path conditions with additional constraints over the features of interest. For instance, the original constraint for EPackage could be extended in several ways to generate different models:

```
-- Several subpackages in a nested package
EPackage.allInstances()→exists(p |
 not p.eSuperPackage.oclIsUndefined() and
 p.eSubPackages→size() > 2)

-- Enforce a value in nsURI value
EPackage.allInstances()→exists(p |
 not p.eSuperPackage.oclIsUndefined() and
 p.nsURI = 'test')
```

Listing 7 Constraints to generate diverse models with nested EPackage elements.

5.2. Extending test cases

The generated test cases can be manually modified, for instance to introduce more meaningful attribute values (e.g., for names) or to consider additional cases. This means that, when we generate a new test case, we might be interested on extending the existing test cases as a way to reuse the knowledge that has been manually introduced. The newly generated test case may replace the extended case or become a new test case on its own.

To achieve this we proceed in two steps. First, we find test cases which are relevant for the coverage goal. Second, we use model completion to generate test cases based on the ones that have been found.

Finding relevant test cases. Given a class of interest c, in this step we identify existing test cases with two properties. First, they must not contain objects of type c. Second, they contain at least one object which can be connected to new new objects of type c. Algorithm 2 is in charge of identifying potential extension points for test cases. Given a class c, the output is a list of tuples $(testcase, object, reference)$ in which the first component is a test case, the second a container object and

the third a reference that can be used to initialize a slot in the container object with an object of type c.

Algorithm 2 Identification of extendable test cases.

Input: test cases in the test suite (*testcases*)
Input: class choosen by the user (c)
Output: list of potential test cases to be extended (testcase, object, reference)

```
// Identify potentially relevant references
links ← empty list
foreach cc in classes of the input meta-models do
    foreach ref in references of cc do
        // Is it possible to set the reference with an object of type c?
        if ref.type = c or ref.type.isSuperTypeOf(c) then
            add (cc, ref) to links
    end
end
// Identify relevant test cases by looking for objects whose class
// is a "container class" (cc)
foreach t in testcases do
    foreach o in objects contained in input models of t do
        class ← o.class
        if class = c then
            continue
        foreach ref such that exists (cc, ref) in links where cc = class do
            add (t, o, ref) to tests cases
        end
    end
end
```

Model completion. The algorithm produces a list of potential test cases to be extended, along with the container and the feature to be extended. Let us suppose that we choose EPackage (which is partially covered in version #2 of the transformation). The second test case in Fig. 4 would be choosen since it already has an object which is extendable by an EPackage, which is the root EPackage through the eSubpackages reference. Thus, we generate the following formula.

```
EPackage.allInstances()→
 select(p | p.eSuperpackage.oclIsUndefined())→
 exists(p | p.eSubpackages→notEmpty())
```

Listing 8 Input for USE MV.

There is an additional practical issue to consider. In USE MV the model completion functionality requires the user to set the object bounds to, at least, the number of objects of each class. This is done automatically by our system. In this case, for instance, we set EPackage = 2 (the original package plus one) and EClass = 1 (the original EClass in the model). If no model can be synthesized (e.g., some constraint cannot be satisfied with these bounds), we apply the heuristic of iteratively increment the number of objects, up to 5. This is the same approach used in AnATLyzer (Cuadrado et al. 2016). As future work we aim at using recent results in model diversity to generate better models (Semeráth & Varró 2018; Burgueño et al. 2019).

6. Inspecting test cases

Each time that we generate a new input model (either from scratch or extending an existing one), our system automatically runs the transformation and generates a new output model. In our process, which is interactive, it is the user's task to check if the new output model is as expected or not. Thus, the output model of a test case has two states:

- *Unreviewed*. The output model has been automatically generated but it has not been reviewed by the developer. The model stays in this state until it is a valid output model (i.e., it is what the developer expects to get obtain from the input model).
- *Commited*. The test case has been reviewed and the user considers that the output model can be promoted to become an expected model. From now on, the test case can be passing or failing. We use EMF Compare to check if the generated output model is the same as the expected model.

The underlying idea is to avoid the burden of creating test models manually. On the one hand, input models are automatically synthesized and, if manual modification is required, it is expected that only a few elements or properties need to be edited. On other hand, the expected models are just created when the transformation is run, and the user only needs to inspect them. To facilitate this process, our tool shows the correspondences between the input and output models, along with which elements of the input meta-model has been dynamically covered (see Fig. 3 and Fig. 6). This is done by instrumenting the transformation.

6.1. Transformation instrumentation

The transformation is instrumented in order to generate a fine-grained model gathering dynamic coverage data. It is implemented as a higher-order transformation which generates an additional output model which records the coverage data. Listing 9 shows an excerpt of the instrumented transformation. Every rule is added an imperative block to create a Record object with the information about the execution of the rule. Then, for every feature access, the recordH helper is executed over the source object. This helper returns the same object and as a side effect generates a Record object to register the execution of this piece of code.

```
module Ecore2Use;
create OUT : Use, COV : COVERAGE from IN : Ecore;

rule package2model {
 from e : Ecore!EPackage
  to c : Use!Model (
   classes <- e.recordH('12:14-12-20',
            'navigation', 'EPackage::eClassifiers', e).eClassifiers
 )
do {
 thisModule.record('9:1-14:2', 'matched-rule', 'package2model',
      Sequence{e}, Sequence{c});
 }
}
```

```
...
rule record(location : String, kind : String, info : String,
  sources : Sequence(OclAny), targets : Sequence(OclAny)) {
  to tgt : COVERAGE!Record (
    location <- location,
    kind <- kind,
    info <- info
  )
do {
  tgt.sources <- sources;
  tgt.targets <- targets;
  tgt;
 }
}

helper context OclAny def: recordH(location : String,
      kind : String, info : String, object : OclAny) : OclAny=
 -- Force a side effect
 let dummy : OclAny = thisModule.recordPath(location, kind, info,
   Sequence {object}, Sequence { })
 in self;
```

Listing 9 Instrumented transformation.

The resulting coverage model is a flat model with Record objects. From this, it is straightforward to reconstruct the coverage information, for instance to show source-target relationships graphically.

7. Tool support

To support the development process described in this paper we have implemented a prototype tool as an Eclipse plug-in integrated into AnATLyzer, which is an IDE for ATL model transformations (Cuadrado et al. 2018a)[6]. To connect USE Model Validator with EMF we use EFinder (Cuadrado & Gogolla 2020) which provides an interface to apply model finding with several OCL variants, including ATL, EMF/OCL and AQL[7].

Figure 6 shows some screenshots of the tool. Model transformations are developed in the regular ATL/AnATLyzer editor (label 1). The results of the static analysis to detect transformation problems can be inspected in the Analysis View (label 2). In the prototype tool that has been implemented for this paper, we have added a *Coverage view* (label 3) which includes information about which classes are covered and by which rules. From this, the synthesis of new test cases can be invoked. The test cases associated to a transformation are maintained in a configuration file, with extension .itrafo. There is a dedicated editor to mantain the configuration file easily. The test suite editor (label 4) shows the test cases and allows the developer to run them and to inspect their state. By double-clicking on a test case the user can jump to the test case inspector (label 5) which shows the results of executing the transformation. A test case is commited (or uncommited) by cliking on the Valid check box. The effect is that the generated output model becomes an expected model and the system copies the output model to the test suite folder, in which the input/expected models are stored. This view also offers some information about the dynamic coverage of the transformation.

[6] http://anatlyzer.github.io
[7] https://github.com/jesusc/efinder

8. Related work

There exists a large variety of works dealing with different aspects of model transformation testing (Selim et al. 2012). In this section we review works related to white-box testing, black-box testing based on contracts and test-driven development methods.

Our approach is based on analysing the transformation rules in order to synthesize relevant test cases. Thus, approaches for white box testing of transformations are very related to our approach. One approach is to use an iterative algorithm based on instantiating the classes in the transformation footprint. To instantiate attribute values representative values can be extracted from the transformation (Fleurey et al. 2004). A similar line of work is presented in (Mottu et al. 2012) in which the transformation footprint is also used to drive test generation, but using Alloy to generate more diverse models. The work in (Küster & Abd-El-Razik 2006) reports the experience of testing several model transformation in the business modelling domain. The authors discuss several types of errors (in our setting, some of them are statically detected by AnATLyzer) and discuss how to use coverage for testing, including an heuristic approach for generating test cases that violate transformation constraints. In our case, we use model finding which may provide more accurate results. ATLTest (González & Cabot 2012) is a white-box test generation tool for ATL. It generates input models trying to maximize the coverage of the transformation. However, this approach does not handle the need of oracles (e.g., valid expected models). Our approach does consider the construction and maintenance of the test suite by doing the generation of input models interactively. An important difference of our proposal with respect to these works is that we try to instantiate models which are not covered yet by the transformation. This is because our approach is interactive and it assumes that the transformation is incomplete. Thus, the coverage criteria is achieved as the transformation is fixed to address the test cases.

An alternative to deriving test cases from the transformation is to use transformation specifications, typically in the form of transformation contracts. Tracts (Gogolla & Vallecillo 2011) are a generalization of transformation contracts, based on establishing relationships between the source and target metamodels using OCL expressions. From this, test cases can be automatically generated. Visual constracts (Guerra, de Lara, Wimmer, et al. 2013) has been proposed to develop transformation contracts. This work is extended in (Guerra & Soeken 2015) to generate input test models and oracle functions.

Tools to analyse and enrich the test suite are also of interest. For instance, there are tools to analyze the coverage of the test suite (Küster et al. 2013) and method to create test cases that increase the quality of the test suite is presented in (Kovács & Küster n.d.). Also, the work in (Gerking et al. 2015) uses a domain-specific language to help in the design of model transformation tests. Classifying terms (Hilken et al. 2018) is a technique which allows the space of possible input models to be partitioned into equivalence classes using OCL expressions, so that more meaningful test cases can be automatically derived using USE Model Validator.

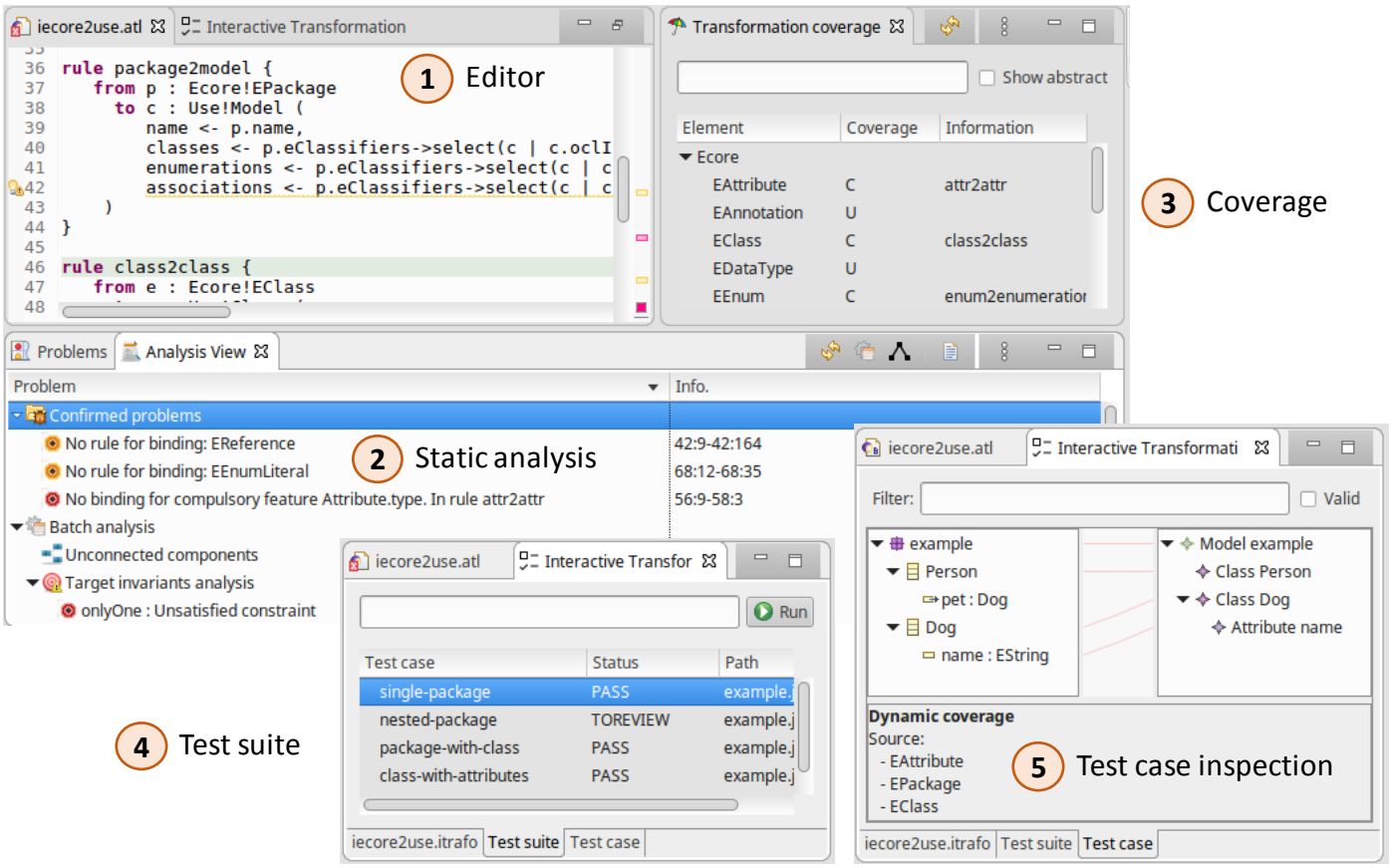

Figure 6 Screenshots of the tool.

Another related line of work is development process for model transformations. The work in (Candel et al. 2019) presents a practical approach for migrating PL/SQL code using model transformations. It proposes a test-driven method in which input models (created by writing pieces of PL/SQL programs) are created manually and output models are inspected to check if they are valid. However, the notion of expected model does not seem to be considered, therefore the process is not automated. In (Küster et al. 2009) a process is proposed to develop model transformation chains using automated testing. It is similar to our process, but the construction of the test cases is fully manual. In our case, the construction is semi-automatic and the output models can be automatically "commited" as test cases by our dedicated tool. The transML (Guerra, de Lara, Kolovos, et al. 2013) development method includes the so-called "transformation cases" which are pairs of concrete source and target models. These cases are transformed into validation code. The creation of these transformation cases can be costly, and our system is intended to reduce the cost of creating them.

9. Conclusions

Model transformation technology still needs from effective testing strategies which are applicable in practice. This paper has presented our initial work to support a test-driven process for model transformation, in which a dedicated tool helps the developer create and mantain the test suite interactively. The tool has been implemented as part of ANATLYZER. Even though the proposed development process is simple, our hope is that its own simplicity helps to increase the use of test cases in the development of ATL transformations. Finally, the paper showcases the use of the capabilities of USE MV to enhance a model transformation development tool.

As future work we plan to investigate how to synthesize more realistic and diverse models (e.g., reusing recent advances (Burgueño et al. 2019; Semeráth & Varró 2018)). This includes being able to generate adequate attribute values according to the domain of the model. Moreover, we would like to carry out an empirical study to understand better the advantages and disadvantages of this approach and to improve the tool accordingly. Another interesting line of work is to try to synthesize rules from the test cases validated by the user, using search-based techniques.

Acknowledgments

I am thankful to Martin Gogolla for being a source of inspiration for all the modelling community. Notably, all the work around USE Model Validator has been particularly inspiring to me, and I am grateful for all the effort that has been put in making USE such a robust and usable tool.

References

Arendt, T., Biermann, E., Jurack, S., Krause, C., & Taentzer, G.

(2010). Henshin: advanced concepts and tools for in-place emf model transformations. In *International conference on model driven engineering languages and systems* (pp. 121–135).

Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., & Mottu, J.-M. (2010). Barriers to systematic model transformation testing. *Communications of the ACM*, *53*(6), 139–143.

Burgueño, L., Cabot, J., Clarisó, R., & Gogolla, M. (2019). A systematic approach to generate diverse instantiations for conceptual schemas. In *International conference on conceptual modeling* (pp. 513–521).

Burgueño, L., Troya, J., Wimmer, M., & Vallecillo, A. (2014). Static fault localization in model transformations. *IEEE Transactions on Software Engineering*, *41*(5), 490–506.

Candel, C. J. F., Molina, J. G., Ruiz, F. J. B., Barceló, J. R. H., Ruiz, D. S., & Viera, B. J. C. (2019). Developing a model-driven reengineering approach for migrating pl/sql triggers to java: A practical experience. *Journal of Systems and Software*, *151*, 38–64.

Cheng, Z., & Tisi, M. (2017). A deductive approach for fault localization in atl model transformations. In *International conference on fundamental approaches to software engineering* (pp. 300–317).

Cuadrado, J. S., & Gogolla, M. (2020). Model finding in the emf ecosystem. In *16th european conference on modelling foundations and applications* (pp. 1–20).

Cuadrado, J. S., Guerra, E., & de Lara, J. (2016). Static analysis of model transformations. *IEEE Transactions on Software Engineering*, *43*(9), 868–897.

Cuadrado, J. S., Guerra, E., & de Lara, J. (2018a). Anatlyzer: an advanced ide for atl model transformations. In *Proceedings of the 40th international conference on software engineering: Companion proceeedings* (pp. 85–88).

Cuadrado, J. S., Guerra, E., & de Lara, J. (2018b). Quick fixing atl transformations with speculative analysis. *Software & Systems Modeling*, *17*(3), 779–813.

Cuadrado, J. S., Guerra, E., de Lara, J., Clarisó, R., & Cabot, J. (2017). Translating target to source constraints in model-to-model transformations. In *2017 acm/ieee 20th international conference on model driven engineering languages and systems (models)* (pp. 12–22).

Cuadrado, J. S., Molina, J. G., & Tortosa, M. M. (2006). Rubytl: A practical, extensible transformation language. In *European conference on model driven architecture-foundations and applications* (pp. 158–172).

Fleurey, F., Steel, J., & Baudry, B. (2004). Validation in model-driven engineering: testing model transformations. In *Proceedings. 2004 first international workshop on model, design and validation, 2004.* (pp. 29–40).

Gerking, C., Ladleif, J., & Schäfer, W. (2015). Model-driven test case design for model-to-model semantics preservation. In *Proceedings of the 6th international workshop on automating test case design, selection and evaluation* (pp. 1–7).

Gogolla, M., Hilken, F., & Doan, K.-H. (2018). Achieving model quality through model validation, verification and exploration. *Computer Languages, Systems & Structures*, *54*, 474–511.

Gogolla, M., & Vallecillo, A. (2011). Tractable model transformation testing. In *European conference on modelling foundations and applications* (pp. 221–235).

González, C. A., & Cabot, J. (2012). Atltest: a white-box test generation approach for atl transformations. In *International conference on model driven engineering languages and systems* (pp. 449–464).

Guana, V., & Stroulia, E. (2014). Chaintracker, a model-transformation trace analysis tool for code-generation environments. In *International conference on theory and practice of model transformations* (pp. 146–153).

Guerra, E., Cuadrado, J. S., & de Lara, J. (2019). Towards effective mutation testing for atl. In *2019 acm/ieee 22nd international conference on model driven engineering languages and systems (models)* (pp. 78–88).

Guerra, E., de Lara, J., Kolovos, D. S., Paige, R. F., & dos Santos, O. M. (2013). Engineering model transformations with transml. *Software & Systems Modeling*, *12*(3), 555–577.

Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., ... Schwinger, W. (2013). Automated verification of model transformations based on visual contracts. *Automated Software Engineering*, *20*(1), 5–46.

Guerra, E., & Soeken, M. (2015). Specification-driven model transformation testing. *Software & Systems Modeling*, *14*(2), 623–644.

Hilken, F., Gogolla, M., Burgueño, L., & Vallecillo, A. (2018). Testing models and model transformations using classifying terms. *Software & Systems Modeling*, *17*(3), 885–912.

Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming*, *72*(1-2), 31–39.

Kolovos, D. S., Paige, R. F., & Polack, F. (2008). The epsilon transformation language. In *Proc. of the 1st international conference in theory and practice of model transformations (icmt'08)* (pp. 46–60). doi: 10.1007/978-3-540-69927-9_4

Kovács, D., & Küster, J. M. (n.d.). A method for creating a test case set to achieve maximum specification coverage in model transformation testing.

Kuhlmann, M., & Gogolla, M. (2012). From uml and ocl to relational logic and back. In *International conference on model driven engineering languages and systems* (pp. 415–431).

Küster, J. M., & Abd-El-Razik, M. (2006). Validation of model transformations–first experiences using a white box approach. In *International conference on model driven engineering languages and systems* (pp. 193–204).

Küster, J. M., Gschwind, T., & Zimmermann, O. (2009). Incremental development of model transformation chains using automated testing. In *International conference on model driven engineering languages and systems* (pp. 733–747).

Küster, J. M., Kovacs, D., Bauer, E., & Gerth, C. (2013). *Integrating coverage analysis into test-driven development of model transformations* (Tech. Rep.). IBM Research Report RZ 3846, IBM Research-Zurich.

Mottu, J.-M., Sen, S., Tisi, M., & Cabot, J. (2012). Static analysis of model transformations for effective test generation. In *2012 ieee 23rd international symposium on software*

reliability engineering (pp. 291–300).
Oakes, B. J., Troya, J., Lúcio, L., & Wimmer, M. (2018). Full contract verification for atl using symbolic execution. *Software & Systems Modeling*, *17*(3), 815–849.
OMG. (2005). Mof qvt final adopted specification (Computer software manual No. ptc/05-11-01). (OMG doc. ptc/05-11-01)
Selim, G. M., Cordy, J. R., & Dingel, J. (2012). Model transformation testing: The state of the art. In *Proceedings of the first workshop on the analysis of model transformations* (pp. 21–26).
Semeráth, O., & Varró, D. (2018). Iterative generation of diverse models for testing specifications of dsl tools. In *Fase* (Vol. 18, pp. 227–245).

About the author

Jesús Sánchez Cuadrado is a Ramón y Cajal researcher at Universidad de Murcia. Earlier he was Assistant Professor at Universidad Autónoma de Madrid. His research has been focused on Model-Driven Engineering, in particular model transformations and Domain-Specific Languages. He has created a number of tools, among others RubyTL, AnATLyzer, EFinder, and http://mar-search.org. They are available at http://github.com/jesusc. You can contact the author at jesusc@um.es or visit http://sanchezcuadrado.es.