Test Data Generation for Model Transformations Combining Partition and Constraint Analysis

Carlos A. González and Jordi Cabot

AtlanMod, École des Mines de Nantes - INRIA, LINA, Nantes, France
{carlos.gonzalez,jordi.cabot}@mines-nantes.fr

Abstract. Model-Driven Engineering (MDE) is a software engineering paradigm where models play a key role. In a MDE-based development process, models are successively transformed into other models and eventually into the final source code by means of a chain of model transformations. Since writing model transformations is an error-prone task, mechanisms to ensure their reliability are greatly needed. One way of achieving this is by means of testing. A challenging aspect when testing model transformations is the generation of adequate input test data. Most existing approaches generate test data following a black-box approach based on some sort of partition analysis that exploits the structural features of the source metamodel of the transformation. However, these analyses pay no attention to the OCL invariants of the metamodel or do it very superficially. In this paper, we propose a mechanism that systematically analyzes OCL constraints in the source metamodel in order to fine-tune this partition analysis and therefore, the generation of input test data. Our mechanism can be used in isolation, or combined with other black-box or white-box test generation approaches.

1 Introduction

Model-Driven Engineering (MDE) is a software engineering paradigm that promotes the utilization of models as primary artifacts in all software engineering activities. When software is developed following a MDE-based approach, models and model transformations are used to (partially) generate the source code for the application to be built.

Writing model transformations is a delicate, cumbersome and error-prone task. In general, MDE-based processes are very sensitive to the introduction of defects. A defect in a model or a model transformation can be easily propagated to the subsequent stages, thus causing the production of faulty software. This is especially true when developing systems of great size and complexity, which usually requires writing large chains of complex model transformations.

In order to alleviate the impact defects can cause, a great deal of effort has been made to find mechanisms and techniques to increase the robustness of MDE-based processes. Thus far, these efforts have been centered on trying to somewhat adapt well-known approaches such as testing or verification to the reality of models and model transformations of MDE (see [1] or [5] for recent

D. Di Ruscio and D. Varró (Eds.): ICMT 2014, LNCS 8568, pp. 25–41, 2014.

surveys). This has resulted in the appearance of a series of testing and verification techniques, specifically designed to target models or model transformations.

In the particular case of testing model transformations, the current picture shares a great deal of similarity with that of traditional testing approaches. Roughly speaking, testing a model transformation consists in first, automatically generating a set of test cases (henceforth test models), second, exercising the model transformation using the generated test models as an input, and finally, checking whether the execution yielded any errors. However, since models are complex structures conforming to a number of constraints defined in a source metamodel, the first and third steps are particularly challenging [4,5].

When addressing test models generation, and along the lines of adapting well-known approaches, expressions such as black-box, white-box or mutation analysis are also of common application. Actually, the black-box paradigm based on the analysis of the model transformation specification is the most exploited one and has given way to a number of techniques (for example [10] or [15]). The objective here is to analyze the model transformation's input metamodel, with the intent of generating a set of test models representative of its instance space, something known as metamodel coverage. The problem though, is that a metamodel's instance space is usually infinite, so what the majority of these methods really do is to use partition analysis to identify non-empty and disjoint regions of the instance space where models share the same features.

The challenge when using partition analysis is building the best partition possible. Since one test model is usually created out of each region identified, partitions should be small enough, so that all the models from the same region are as homogeneous as possible (meaning that the sample model from that region can be used to represent all models from that same region and reduce, this way, the number of test models to use to get a sufficient confidence level on the quality of the transformation). Existing approaches address this by taking advantage of the fact that input metamodels usually come in the form of UML class diagrams complemented with constraints expressed in the OCL (Object Constraint Language). Therefore, partition analysis focuses on elements like association multiplicities, attributes values or OCL constraints to partition the model. However, in this last case, current approaches tend to be very superficial, either focusing only on simple OCL constraints, or deriving just obvious regions that do not require a deep analysis. This limits the representativeness of the generated test models and also the degree of coverage achieved when dealing with non-trivial metamodels.

In this paper, we propose a mechanism for the generation of input test models based on a combination of constraint and partition analysis over the OCL invariants of the model transformation's input metamodel. The method covers a substantial amount of OCL constructs and offers up to three different test model generation modes. Besides, it can be used in isolation, or combined with other black-box or white-box approaches to enhance the testing experience.

The paper is organized as follows: Section 2 outlines our proposal. Section 3 focuses on the analysis of OCL invariants to identify suitable regions of the instance

space. Section 4 describes the three test model generation modes. Section 5 is about the implementation of the approach and some scenarios where the tool could be useful. Section 6 reviews the related work and finally, in Section 7, we draw some conclusions and outline the future work.

2 Overview of Our Approach

Category-partition testing [16] consists in partitioning the input domain of the element under test, and then selecting test data from each class in the partition. The rationale here is that, for the purpose of testing, any element of a class is as good as any other when trying to expose errors.

According to this philosophy, our approach is depicted in Fig. 1. The model transformation's input metamodel characterizes a certain domain, and its instance space, possible inputs for the transformation. In the figure, dashed arrows indicate what characterizes certain elements, whereas solid arrows are data flows. When generating test models, the component called "OCL Analyzer" partitions the metamodel's instance space by analyzing its OCL invariants (Sections 3 and 4). As a result, a series of new OCL invariants characterizing the regions of the partition are obtained. This information, along with the input metamodel is then given to the "Test Model Generator" component, for the actual creation of the test models (Section 4).

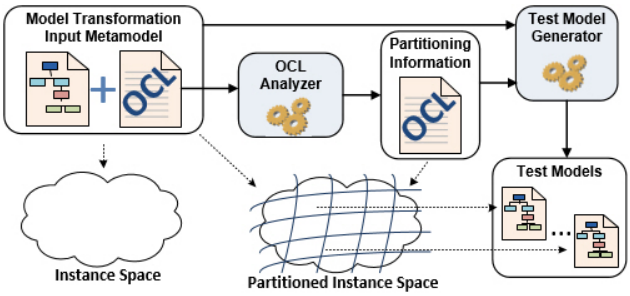

Fig. 1. Overall picture

As mentioned before, the main difference between our approach and other black-box ones based on partition analysis is the way OCL invariants are analyzed. Whereas our approach is capable of analyzing the majority of OCL constructs in a systematic way, approaches like [10] or [15] build partitions by exploiting only simple OCL expressions that explicitly constraint the values a given model element can take. This is because numeric or logical values are an easy target at the time of identifying regions in the instance space. In what follows, we compare this type of analysis with our proposal to show that they are, in many cases, insufficient to derive representative test models.

Fig. 2(a) shows a metamodel describing the relationship between research teams and the papers they submit for publication. A simple partition analysis

would try to exploit the presence of a numerical value in the OCL invariant stating that every team must have more than 10 submissions accepted. However, that alone is not enough to generate an interesting partitioning. A more fine-grained analysis of the constraint would reveal that beyond testing the transformation with teams with more than 10 accepted submissions, you should also test the transformation with teams with more than 10 accepted submissions and at least one rejected one. Our method reaches this conclusion by analyzing the "select" condition in the OCL expression (more details on this later on). Fig. 3 shows the difference in the output produced by both analyses. Obviously, the second one exercises more the transformation and therefore may uncover errors not detected when using only the first one.

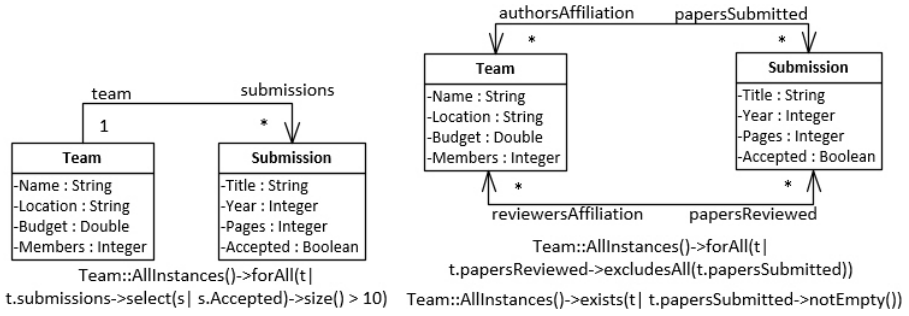

Fig. 2. Metamodels of the examples used throughout the paper

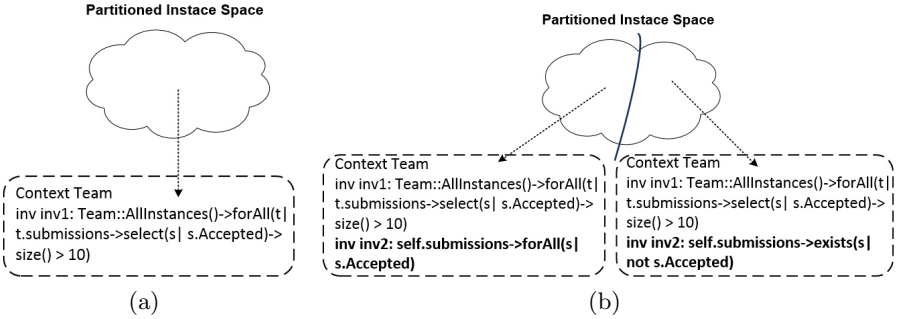

Fig. 3. Results of two different partition analyses over the metamodel example

3 OCL Analysis

In this section, we begin the description of how to identify partitions in the input metamodel's instance space, focusing on the first step: analyzing the OCL invariants in the input metamodel to generate new OCL invariants characterizing suitable regions of the instance space. Next section uses these constraints to create the actual partitions.

Firstly, we talk about the OCL constructs supported by the method. After that, we describe how to systematically analyze complex OCL invariants made up by arbitrary combinations of the supported constructs.

3.1 OCL Constructs Supported

The supported OCL constructs have been classified in five groups and presented here in tabular form. The first group corresponds to expressions involving the presence of boolean operators (Table 1). The second group is about expressions formed by a boolean function operating over the elements of a collection (Table 2). The third group includes those boolean expressions involving the presence of arithmetic operators (Table 3). The fourth group contains other non-boolean expressions, that can be part of more complex boolean expressions (Table 4). Finally, the last group (Table 5) shows equivalent expressions for boolean expressions from Tables 1, 2 and 3 when they are negated.

Tables 1, 2, 3 and 4 share the same structure. For any given row, the second column contains a pattern. Analyzing an OCL invariant implies looking for these patterns, and every time one of them matches, the information in the third column indicates how to derive new OCL expressions characterizing suitable regions in the instance space. A dash (-) indicates that no new OCL expressions are derived. The rationale behind a given pattern and the expressions in the "Regions" column is simple: the pattern represents the invariant that the model must hold, and the information in the "Regions" column are more refined expressions that must also hold when the pattern holds. For example, the entry 1 in Table 1 indicates that the pattern expression holds if the two subexpressions evaluate to the same value. The subexpressions in the "Regions" column indicate that there are two possibilities for this: either both are true, or both are false.

Table 5 is slightly different, though, and that has to do with how the method deals with negated expressions. Each time a negated expression is found, it must be substituted by an equivalent non-negated expression before any new regions can be identified. Second column in the table shows boolean expressions from Tables 1, 2 and 3. The third column contains the equivalents to these expressions when they are negated. In some cases, the substitution process must be applied recursively since, for some expressions, the negated equivalent can also contain negated subexpressions.

3.2 Analyzing OCL expressions

Typically, real-life OCL invariants will be composed by combinations of some of the patterns described above. In the following we give the intuition of how to process some of these combined expressions, in particular, those of type $source \rightarrow operation(argument)$[1]:

[1] For the case of more complex expressions, involving boolean (AND, OR, ...) or logical operators ($\leq$, $>$, ...), the process is quite the same. However, this full process cannot be described here due to lack of space.

Table 1. Expressions Involving Boolean Operators

	Pattern	Regions
1	$BExp_1 = BExp_2$	$BExp_1 = FALSE\ AND\ BExp_2 = FALSE$ $BExp_1 = TRUE\ AND\ BExp_2 = TRUE$
2	$BExp_1\ AND\ BExp_2$	$BExp_1 = TRUE\ AND\ BExp_2 = TRUE$
3	$BExp_1\ OR\ BExp_2$	$BExp_1 = FALSE\ AND\ BExp_2 = TRUE$ $BExp_1 = TRUE\ AND\ BExp_2 = FALSE$ $BExp_1 = TRUE\ AND\ BExp_2 = TRUE$
4	$BExp_1\ XOR\ BExp_2$	$BExp_1 = FALSE\ AND\ BExp_2 = TRUE$ $BExp_1 = TRUE\ AND\ BExp_2 = FALSE$
5	$BExp_1 <> BExp_2$	$BExp_1 = TRUE\ AND\ BExp_2 = FALSE$ $BExp_1 = FALSE\ AND\ BExp_2 = TRUE$
6	$Class.BAttr = TRUE$	$Class :: AllInstances() \rightarrow forAll(c\mid c.BAttr = TRUE)$
7	$Class.BAttr = FALSE$	$Class :: AllInstances() \rightarrow forAll(c\mid c.BAttr = FALSE)$

Table 2. Expressions Featuring Boolean Functions in the Context of a Collection

	Pattern	Regions
1	$col \rightarrow exists(body)$	$col \rightarrow forAll(body)$ $col \rightarrow exists(NOT\ body)$
2	$col \rightarrow one(body)$	$col \rightarrow size() = 1$ $col \rightarrow size() > 1$
3	$col \rightarrow forAll(body)$	$col \rightarrow isEmpty()$ $col \rightarrow notEmpty()$
4	$col \rightarrow includes(o)$	$col \rightarrow count(o) = 1$ $col \rightarrow count(o) > 1$
5	$col \rightarrow excludes(o)$	$col \rightarrow isEmpty()$ $col \rightarrow notEmpty()$
6	$col_1 \rightarrow includesAll(col_2)$	$col_1 \rightarrow size() = col_2 \rightarrow size()$ $col_1 \rightarrow size() > col_2 \rightarrow size()$
7	$col_1 \rightarrow excludesAll(col_2)$	$col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow isEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow isEmpty()$
8	$col \rightarrow isEmpty()$	–
9	$col \rightarrow notEmpty()$	–

1. Find a pattern matching the whole invariant. If not found, end here.
2. Generate the new OCL expressions corresponding to the pattern matched.
3. Find a pattern matching the "source" expression.
4. If found, generate the OCL expressions corresponding to the pattern matched.
5. Repeat the process recursively over the subexpressions in the "source" expression, until no more matchings are found.
6. Find a pattern matching the "argument" expression.
7. If found, generate the OCL expressions corresponding to the pattern matched.
8. Repeat the process recursively over the subexpressions in the "argument" expression, until no more matchings are found.

Table 3. Boolean Expressions Involving Arithmetic Operators

	Pattern	**Regions**
1	$col_1 \rightarrow size() = col_2 \rightarrow size()$	$col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow isEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$
2	$col_1 \rightarrow size() = NUM$	–
3	$col_1 \rightarrow size() <> col_2 \rightarrow size()$	$col_1 \rightarrow size() > col_2 \rightarrow size()\ AND$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow size() < col_2 \rightarrow size()\ AND$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow isEmpty()$
4	$col \rightarrow size() <> NUM\ AND$ $NUM <> 0$	$col \rightarrow size() > NUM$ $col \rightarrow notEmpty()\ AND\ col \rightarrow size() < NUM$ $col \rightarrow isEmpty()$
5	$col_1 \rightarrow size() >= col_2 \rightarrow size()$	$col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow isEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow isEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$
6	$col \rightarrow size() >= NUM$	$col \rightarrow size() > NUM$ $col \rightarrow size() = NUM$
7	$col_1 \rightarrow size() > col_2 \rightarrow size()$	$col_2 \rightarrow isEmpty()$ $col_2 \rightarrow notEmpty()$
8	$col \rightarrow size() > NUM$	–
9	$col_1 \rightarrow size() <= col_2 \rightarrow size()$	$col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow isEmpty()$ $col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$
10	$col \rightarrow size() <= NUM\ AND$ $NUM <> 0$	$col \rightarrow size() < NUM$ $col \rightarrow size() = NUM$ $col \rightarrow isEmpty()$
11	$col_1 \rightarrow size() < col_2 \rightarrow size()$	$col_1 \rightarrow isEmpty()$ $col_1 \rightarrow notEmpty()$
12	$col \rightarrow size() < NUM$	$col \rightarrow isEmpty()$ $col \rightarrow notEmpty()$
13	$col \rightarrow count(o) > NUM$	$col \rightarrow excluding(o) \rightarrow isEmpty()$ $col \rightarrow excluding(o) \rightarrow notEmpty()$
14	$col \rightarrow count(o) = NUM$	$col \rightarrow excluding(o) \rightarrow isEmpty()$ $col \rightarrow excluding(o) \rightarrow notEmpty()$
15	$col \rightarrow count(o) < NUM$	$col \rightarrow isEmpty()$ $col \rightarrow notEmpty()\ AND$ $col \rightarrow excluding(o) \rightarrow notEmpty()$ $col \rightarrow notEmpty()\ AND$ $col \rightarrow excluding(o) \rightarrow isEmpty()$
16	$Class.NumAttr > NUM$	$Class :: AllInstances() \rightarrow$ $forAll(c\mid c.NumAttr > NUM)$
17	$Class.NumAttr < NUM$	$Class :: AllInstances() \rightarrow$ $forAll(c\mid c.NumAttr < NUM)$
18	$Class.NumAttr = NUM$	$Class :: AllInstances() \rightarrow$ $forAll(c\mid c.NumAttr = NUM)$

Table 4. Other OCL Functions

	Pattern	**Regions**
1	$col \rightarrow select(body)$	$col \rightarrow forAll(body)$ $col \rightarrow exists(NOT\ body)$
2	$col \rightarrow reject(body)$	$col \rightarrow forAll(NOT\ body)$ $col \rightarrow exists(body)$
3	$col \rightarrow collect(body)\ AND$ $body.oclIsTypeOf(boolean)$	$col \rightarrow forAll(body)$ $col \rightarrow exists(NOT\ body)$
4	$col_1 \rightarrow union(col_2)$	$col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow isEmpty()$ $col_1 \rightarrow isEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow notEmpty()$ $col_1 \rightarrow notEmpty()\ AND\ col_2 \rightarrow isEmpty()$
5	$col_1 \rightarrow intersection(col_2)$	$col_1 = col_2$ $col_1 \rightarrow includesAll(col_2)\ AND$ $\quad col_1 \rightarrow size() > col_2 \rightarrow size()$ $col_2 \rightarrow includesAll(col_1)\ AND$ $\quad col_2 \rightarrow size() > col_1 \rightarrow size()$ $col_1 <> col_2$
6	$col \rightarrow excluding(o)$	$col \rightarrow isEmpty()$ $col \rightarrow notEmpty()$
7	$col \rightarrow subsequence(l, u)$	$col \rightarrow size() = u - l$ $col \rightarrow size() > u - l$
8	$col \rightarrow at(n)$	$col \rightarrow size() = n$ $col \rightarrow size() > n$
9	$col \rightarrow any(body)$	$col \rightarrow forAll(body)$ $col \rightarrow exists(NOT\ body)$

9. Once the matching phase finishes, every constraint from each matching group is AND-combined with each one in the rest of the groups. This way, the final list of OCL expressions is obtained. Each of these OCL expressions characterizes a region of the input metamodel's instance space.

As an example, Fig. 2(b) shows another version of the metamodel describing the relationship between research teams and the papers they submit. It includes two OCL invariants. The first one states that the members of a team do not review their own papers, and the second one says that at least one of the teams must have at least one submission.

The analysis starts with the first invariant. It features a "forAll" operation matching entry 3 in Table 2. That entry says that the instance space can be divided in two regions. The region of models with no teams, and the one of models with any number of teams except zero. They can be characterized as:

Team::AllInstances()–>isEmpty() (A1.1)
Team::AllInstances()–>notEmpty() (A1.2)

Now, a pattern matching the "argument" of the "forAll" operation is searched. Entry 6 in Table 2 matches. Since the expression is embedded as the argument of a higher level operator, its context must be identified to build the new OCL expressions properly. By doing this, the following OCL constraints are obtained:

Table 5. Boolean Expressions And Their Negated Equivalents

	Pattern	**Negated Equivalent**
1	$BExp_1 = BExp_2$	$BExp_1 <> BExp_2$
2	$BExp_1\ AND\ BExp_2$	$NOT\ BExp_1\ OR\ NOT\ BExp_2$
3	$BExp_1\ OR\ BExp_2$	$NOT\ BExp_1\ AND\ NOT\ BExp_2$
4	$BExp_1\ XOR\ BExp_2$	$BExp_1 = BExp_2$
5	$col_1 \rightarrow exists(body)$	$col_1 \rightarrow forAll(NOT\ body)$
6	$col_1 \rightarrow one(body)$	$col_1 \rightarrow select(body) \rightarrow size() <> 1$
7	$col_1 \rightarrow forAll(body)$	$col_1 \rightarrow exists(NOT\ body)$
8	$col_1 \rightarrow includes(o)$	$col_1 \rightarrow excludes(o)$
9	$col_1 \rightarrow isEmpty()$	$col_1 \rightarrow notEmpty()$
10	$col_1 \rightarrow size() = col_2 \rightarrow size()$	$col_1 \rightarrow size() <> col_2 \rightarrow size()$
11	$col_1 \rightarrow size() > col_2 \rightarrow size()$	$col_1 \rightarrow size() \leq col_2 \rightarrow size()$
12	$col_1 \rightarrow size() < col_2 \rightarrow size()$	$col_1 \rightarrow size() \geq col_2 \rightarrow size()$
13	$col \rightarrow size() \leq NUM\ AND\ NUM <> 0$	$col \rightarrow size() > NUM$
14	$col \rightarrow size() <> NUM\ AND\ NUM <> 0$	$col \rightarrow size() = NUM$
15	$col \rightarrow size() = NUM$	$(col \rightarrow size() > NUM)\ OR$ $(col \rightarrow size() < NUM)$
16	$col \rightarrow size() > NUM$	$(col \rightarrow size() = NUM)\ OR$ $(col \rightarrow size() < NUM)$
17	$col \rightarrow count(o) > NUM$	$(col \rightarrow count(o) < NUM)\ OR$ $(col \rightarrow count(o) = NUM)$
18	$col \rightarrow count(o) = NUM$	$(col \rightarrow count(o) < NUM)\ OR$ $(col \rightarrow count(o) > NUM)$
19	$col \rightarrow count(o) < NUM$	$(col \rightarrow count(o) = NUM)\ OR$ $(col \rightarrow count(o) > NUM)$
20	$Class.NumAttr > NUM$	$(Class.NumAttr < NUM)\ OR$ $(Class.NumAttr = NUM)$
21	$Class.NumAttr < NUM$	$(Class.NumAttr > NUM)\ OR$ $(Class.NumAttr = NUM)$
22	$Class.NumAttr = NUM$	$(Class.NumAttr < NUM)\ OR$ $(Class.NumAttr > NUM)$

```
Team::AllInstances()–>forAll(t|t.papersReviewed–>isEmpty()
  and t.papersSubmitted–>NotEmpty())                    (A2.1)
Team::AllInstances()–>forAll(t|t.papersReviewed–>isEmpty()
  and t.papersSubmitted–>isEmpty())                     (A2.2)
Team::AllInstances()–>forAll(t|t.papersReviewed–>NotEmpty()
  and t.papersSubmitted–>NotEmpty())                    (A2.3)
Team::AllInstances()–>forAll(t|t.papersReviewed–>NotEmpty()
  and t.papersSubmitted–>isEmpty())                     (A2.4)
```

With this, the matching phase over the first invariant is over. The rest of elements in the invariant do not match any pattern. Now, the resulting two groups (A1.X and A2.X) must be combined. This produces the following list of expressions:

Team::AllInstances()–>isEmpty() and Team::AllInstances()–>forAll(t|
t.papersReviewed–>isEmpty() and t.papersSubmitted–>NotEmpty()) (A3.1)
Team::AllInstances()–>isEmpty() and Team::AllInstances()–>forAll(t|
t.papersReviewed–>isEmpty() and t.papersSubmitted–>isEmpty()) (A3.2)
Team::AllInstances()–>isEmpty() and Team::AllInstances()–>forAll(t|
t.papersReviewed–>NotEmpty() and t.papersSubmitted–>NotEmpty()) (A3.3)
Team::AllInstances()–>isEmpty() and Team::AllInstances()–>forAll(t|
t.papersReviewed–>NotEmpty() and t.papersSubmitted–>isEmpty()) (A3.4)
Team::AllInstances()–>notEmpty() and Team::AllInstances()–>forAll(t|
t.papersReviewed–>isEmpty() and t.papersSubmitted–>NotEmpty()) (A3.5)
Team::AllInstances()–>notEmpty() and Team::AllInstances()–>forAll(t|
t.papersReviewed–>isEmpty() and t.papersSubmitted–>isEmpty()) (A3.6)
Team::AllInstances()–>notEmpty() and Team::AllInstances()–>forAll(t|
t.papersReviewed–>NotEmpty() and t.papersSubmitted–>NotEmpty()) (A3.7)
Team::AllInstances()–>notEmpty() and Team::AllInstances()–>forAll(t|
t.papersReviewed–>NotEmpty() and t.papersSubmitted–>isEmpty()) (A3.8)

With this, the analysis of the first invariant is finished. The analysis of the second invariant is analogous and yields the constraints in the group B1.X.

Team::AllInstances()–>forAll(t|t.papersSubmitted–>notEmpty()) (B1.1)
Team::AllInstances()–>exists(t|not t.papersSubmitted–>notEmpty()) (B1.2)

Putting all together, the analysis of the two invariants in the model of Fig. 2(b) yielded the groups of constraints A3.X and B1.X, respectively. Each constraint in these groups characterizes a region of the instance space. They will be the input for the test model generation phase, described in the next section.

Finally, it is important to mention that the analysis of OCL invariants is not free from inconveniences. From the example, it can be easily seen that some of the generated constraints could be simplified (for example in A.3.1, if there are no "Team" instances, then there is no need to check the subexpression at the right of "and"). More importantly, some of the constraints produced in the combination stage could be inconsistent. These problems can be addressed in two different ways: adding a post-processing stage at this point to "clean" the constraints obtained, or addressing them directly during the test model creation stage (our preferred alternative, as we explain in the next section).

4 Partition Identification and Test Models Generation

This section details the identification of partitions and the generation of test models from the sets of constraints obtained in the previous step. Our approach provides three different alternatives depending on the effort the tester wants to invest to ensure the absence of overlapping test models.

4.1 Simple Mode

As shown before, the analysis of one OCL invariant yields a list of new OCL expressions, each one characterizing a region of the instance space. It cannot

be guaranteed though, that these regions do not overlap (i.e. that they constitute a partition). Looking back at the example, this means that the regions in A3.X might overlap, and the same goes for the regions in B1.X (we have two groups here because we had analyzed two invariants). Fig. 4(a) and 4(b) illustrate the best- and worst-case scenarios when three regions are identified from the analysis of a given invariant. In the worst case, a generated test model to cover, for example, region 4, could indeed "fall into" this area, or in any of the adjacent overlapping areas labeled with a question mark (?). In this situation, when regions overlap, it is likely that generated test models do it as well.

Ensuring that a number of regions do not overlap requires additional effort, but in "Single Mode", no further effort to identify partitions is made. It simply runs the test model generator over the regions that were identified in the OCL analysis, each time passing the input metamodel (and its OCL invariants), and one of the OCL expressions characterizing these regions. It represents a cheaper way (compared to the other alternatives) of creating test models without ensuring that they will not overlap. Running "Single Mode" over the example of Fig. 2(b) consists in invoking the model generator for each of the OCL expressions in A3.X and B1.X.

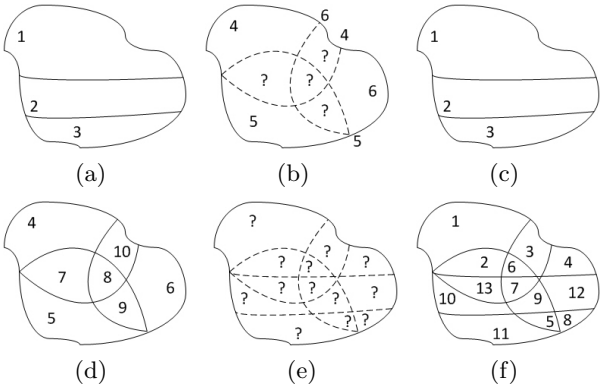

Fig. 4. Overlapping and partitions when generating test models

4.2 Multiple-Partition Mode

Given the set of OCL expressions obtained from the analysis of one OCL invariant, "Multiple-Partition Mode" produces a new set of OCL expressions that constitute a partition (i.e. do not overlap each other) of the instance space.

In general, if the analysis of one OCL invariant yields "n" regions, a partition can be derived, with a number of regions somewhere in the interval [n, 2^n - 1]. Although the exact number depends on how the original "n" regions overlap each other, justifying the lower and upper bounds is rather simple. To show this, we will focus on the particular case of $n = 3$ and refer to the OCL expressions characterizing these regions as $B_i, i = 1..3$.

The lower bound corresponds to the best-case scenario (Fig. 4(a)) where the original "n" regions do already constitute a partition. The upper bound corresponds

to the worst-case scenario (Fig. 4(b)) where the "n" regions overlap each other. In this case, it is possible to derive a partition (Fig. 4(d)) with 7 regions, characterized by the following OCL expressions:

- $D_4 = B_4\ AND\ NOT\ B_5\ AND\ NOT\ B_6$
- $D_5 = B_5\ AND\ NOT\ B_4\ AND\ NOT\ B_6$
- $D_6 = B_6\ AND\ NOT\ B_4\ AND\ NOT\ B_5$
- $D_7 = B_4\ AND\ B_5\ AND\ NOT\ B_6$
- $D_8 = B_4\ AND\ B_5\ AND\ B_6$
- $D_9 = NOT\ B_4\ AND\ B_5\ AND\ B_6$
- $D_{10} = B_4\ AND\ NOT\ B_5\ AND\ B_6$

That is, all the combinations of three elements (the initial number of regions) that can take two different states (to overlap, not to overlap), excepting:

- $NOT\ B_5\ AND\ NOT\ B_4\ AND\ NOT\ B_6$

which is not representative of any region, since it falls out of the instance space. Generalizing for the case of "n" regions, the upper limit of 2^n - 1 is obtained.

Running "Multiple-Partition Mode" over the example of Fig. 2(b) consists in first, creating all the combinations of the OCL expressions in the groups A3.X and B1.X, and then invoking the model generator to process each of them. The combination of the expressions in A3.X yields a list of 255 new expressions, so only the results of combining the OCL expressions in B1.X are shown.

```
Team::AllInstances()–>forAll(t|t.papersSubmitted–>notEmpty()) and
  Team::AllInstances()–>exists(t|not t.papersSubmitted–>notEmpty())          (B2.1)
not Team::AllInstances()–>forAll(t|t.papersSubmitted–>notEmpty()) and
  Team::AllInstances()–>exists(t|not t.papersSubmitted–>notEmpty())          (B2.2)
Team::AllInstances()–>forAll(t|t.papersSubmitted–>notEmpty()) and not
  Team::AllInstances()–>exists(t|not t.papersSubmitted–>notEmpty())          (B2.3)
```

4.3 Unique-Partition Mode

Applying "Multiple-Partition Mode" guarantees that the regions obtained for each OCL invariant do not overlap each other. However, if the input metamodel has more than one invariant, regions in the partition for one invariant might overlap regions in the partitions of the rest of invariants. "Unique-Partition Mode" guarantees that regions do not overlap each other, no matter where they come from. Therefore, in "Unique-Partition Mode" only one partition is characterized, regardless of the number of OCL invariants of the input metamodel. This can be easily seen with an example. If Fig. 4(c) and Fig. 4(d) were the partitions produced by "Multiple-Partition Mode" for two invariants, when putting together, they would overlap as shown in Fig. 4(e). In this scenario "Unique-Partition Mode" would yield the partition of Fig. 4(f).

Applying "Unique-Partition Mode" is a simple three-step process: First, "Multiple-Partition Mode" is applied over each invariant. After that, the lists of OCL expressions characterizing the regions in each partition are merged together to form one big list. Finally "Multiple-Partition Mode" is applied over

that list, to generate the final partition. Applying this mode over the example of Fig. 2(b) consists in merging the results of "Multiple-Partition Mode" shown before ($255 + 3 = 258$ OCL expressions) into one big list and run another iteration of "Multiple-Partition Mode" over that list. Clearly, the main problem for the practical utilization of this approach could be the combinatorial explosion in the number of regions conforming the final partition.

4.4 Creating Test Models

After having described how partitions are generated, the last step is the creation of the actual test models. Without regard of the generation mode selected, this is a pretty straightforward process. When fed with the input metamodel (and its OCL invariants) and an OCL invariant characterizing one region of the input space, the "Test Model Generator" component (Fig. 1) tries to build a valid instance of the input metamodel, that also satisfies this additional OCL constraint. The whole set of test models is obtained by repeating this process as many times as regions were found.

In practical terms, we use a separate tool called EMFtoCSP[2] for that. This tool is capable of looking for valid instances of a given metamodel enriched or not with OCL constraints. One of its nicest features is that it transforms the problem of finding a valid instance into a Constraint Satisfaction Problem (CSP). This is especially convenient to address the issues mentioned at the end of Section 3. For example, when presented with an infeasible combination of constraints, EMFtoCSP can dismiss it, yielding no test model.

5 Implementation and Usage Scenarios

We have implemented an Eclipse[3]-based tool that can generate test models following any of the three generation modes exposed before. It can be downloaded from http://code.google.com/a/eclipselabs.org/p/oclbbtesting/ where the user will find all the necessary information for its installation and usage.

When used in isolation, the tool produces models to cover the instance space of the transformation's input metamodel, out of the OCL invariants of that metamodel. Since graphical constraints in a model, like associations, multiplicities, etc can also be expressed in the form of OCL invariants, as detailed in [11], the tool could also be used to derive test models out of these graphical constraints.

There may be occasions though, in which it is convenient to focus only on specific sections of the input metamodel: the model transformation could only "exercise" a part of the input metamodel, or the tester could only be interested on a specific part of the transformation. In the first case, the tool could be combined with approaches capable of identifying what the relevant sections of the input metamodel are, like for example [10]. In the second case, if the preconditions that trigger specific parts of the model transformation are expressed in such a

[2] `http://code.google.com/a/eclipselabs.org/p/emftocsp/`
[3] `http://www.eclipse.org`

way, that new OCL invariants in the context of the input metamodel can be derived, then these new invariants could be used to limit the generation of test models to those regions of the instance space triggering the sections of the model transformation that are of interest. This could be exploited even further, to allow the generation of test models aimed at satisfying different coverage criteria over the transformation [13].

Finally, the tool could also be useful to complement others that lack the ability to generate test models out of OCL invariants, or do it in a limited way.

6 Related Work

Although not related to model transformation testing, to the best of our knowledge, the first attempt of using partition analysis to derive test models out of UML class diagrams was made by Andrews et al. [3]. In this work, partition analysis is employed to identify representative values of attributes and association ends multiplicities to steer the generation of test models. However, OCL invariants are analyzed only in the context of how they restrict the values an individual attribute can take. This represents only a portion of the analysis of OCL invariants presented in this paper. Andrews et al. served as inspiration for the black-box test model generation approach proposed by Fleurey et al. [10,7] where the partition analysis of [3] is used to identify representative values of the model transformation input metamodel.

The work of Fleurey et al. influenced a number of proposals in this field as well. Lamari [15] proposed a tool for the generation of the effective metamodel out of the specification of a model transformation. Wang et al. [19] proposed a tool for the automatic generation of test cases, by deriving the effective metamodel and representative values out of model transformations rules. Sen et al. [17] presented a tool called "Cartier" for the generation of test cases based on the resolution of a SAT problem by means of Alloy[4]. The SAT problem is built, among other data, out of some model fragments obtained out of a partition analysis of the input metamodel. Since these works are more/less based on the partition analysis technique proposed in [3] the comments made there apply here as well.

Also based on the utilization of constraints solvers are the works of Fiorentini et al. [9] and Guerra [13]. In [9], a logic encoding of metamodels expressed in the MOF[5] language is proposed. The encoding is then exploited by means of a constraint solver, although OCL does not seem to be supported. [13] presents a framework for specification-driven testing, that can be used to generate a complete test suite. It works by transforming invariants and preconditions from the model transformation specification into OCL expressions, that are then fed to a constraint solver.

To finish with black-box approaches, Vallecillo et al. [18] presented a proposal based on the concept of Tract (a generalization of the concept of model transformation contract [4,8]), where test models are generated by means of a language

[4] `http://alloy.mit.edu/alloy/`
[5] `http://www.omg.org/spec/MOF/`

called ASSL, part of the USE tool[6]. In this approach, the characteristics of the test models to be generated, seem to be explicitly indicated beforehand in the ASSL scripts, whereas in our approach that information is derived automatically from the analysis of the OCL invariants of the input metamodel.

Compared to the number of black-box test model generation proposals, the number of existing white-box approaches is rather small. Fleurey et al. [10] complemented their black-box approach by proposing the utilization of the transformation definition to identify relevant values and the effective metamodel, although not mention of OCL is made. Küster et al. [14] proposed three different test model generation techniques following a white-box approach, although an automatic way of building test models out of OCL constraints is not included. Finally, the approach more similar to our work is [12], where test models are characterized by a series of OCL constraints obtained out of the analysis of the model transformation internals.

Finally, test case generation through partition analysis, has also been object of study in the area of model-based testing. Examples of this are [20,6,2].

7 Conclusions

The generation of test models by means of black-box approaches based on partition analysis has largely ignored the valuable information in the OCL constraints. This limits the test generation process and consequently, the degree of coverage achieved over the input metamodel. In this paper, we have presented a black-box test model generation approach for model transformation testing, based on a deep analysis of the OCL invariants in the input metamodel of the transformation. Our method can be configured to be used at three different levels of exhaustiveness, depending on the user's needs. A tool supporting the process has been implemented, and it can be used in isolation or combined with other test model generation approaches. It can also be useful to generate test models at different degrees of coverage.

In the future, we want to expand our method so that it could be used not only for model transformation testing (where all input models are always assumed to be valid metamodel instances) but also for faulty testing (i.e. to test software implementations that should be able to deal appropriately with wrong models). Additionally, we would also like to improve the way OCL expressions characterizing regions of the instance space are generated, to reduce the number of spurious or infeasible combinations produced.

References

1. Ab Rahim, L., Whittle, J.: A survey of approaches for verifying model transformations. Software and System Modeling (June 2013) (Published online)
2. Ali, S., Iqbal, M.Z.Z., Arcuri, A., Briand, L.C.: Generating test data from OCL constraints with search techniques. IEEE Transactions on Software Engineering 39(10), 1376–1402 (2013)

[6] `http://sourceforge.net/projects/useocl/`

3. Andrews, A.A., France, R.B., Ghosh, S., Craig, G.: Test adequacy criteria for UML design models. Software Testing, Verification and Reliability 13(2), 95–127 (2003)
4. Baudry, B., Dinh-Trong, T., Mottu, J.M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Traon, Y.L.: Model transformation testing challenges. In: ECMDA Workshop on Integration of Model Driven Development and Model Driven Testing (2006)
5. Baudry, B., Ghosh, S., Fleurey, F., France, R.B., Traon, Y.L., Mottu, J.M.: Barriers to systematic model transformation testing. Comm. of the ACM 53(6), 139–143 (2010)
6. Bernard, E., Bouquet, F., Charbonnier, A., Legeard, B., Peureux, F., Utting, M., Torreborre, E.: Model-based testing from UML models. In: Informatik 2006. LNI, vol. 94, pp. 223–230. GI (2006)
7. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: An algorithm and a tool. In: 17th Int. Symposium on Software Reliability Engineering, ISSRE 2006, pp. 85–94. IEEE (2006)
8. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: OCL and Model Driven Engineering Workshop (2004)
9. Fiorentini, C., Momigliano, A., Ornaghi, M., Poernomo, I.: A constructive approach to testing model transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 77–92. Springer, Heidelberg (2010)
10. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: Testing model transformations. In: 1st Int. Workshop on Model, Design and Validation, pp. 29–40 (2004)
11. Gogolla, M., Richters, M.: Expressing UML class diagrams properties with OCL. In: Clark, A., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 85–114. Springer, Heidelberg (2002)
12. González, C.A., Cabot, J.: ATLTest: A white-box test generation approach for atl transformations. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 449–464. Springer, Heidelberg (2012)
13. Guerra, E.: Specification-driven test generation for model transformations. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 40–55. Springer, Heidelberg (2012)
14. Küster, J.M., Abd-El-Razik, M.: Validation of model transformations - first experiences using a white box approach. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 193–204. Springer, Heidelberg (2007)
15. Lamari, M.: Towards an automated test generation for the verification of model transformations. In: ACM Symposium on Applied Computing (SAC), pp. 998–1005. ACM (2007)
16. Ostrand, T.J., Balcer, M.J.: The category-partition method for specifying and generating functional tests. Comm. of the ACM 31(6), 676–686 (1988)
17. Sen, S., Baudry, B., Mottu, J.M.: On combining multi-formalism knowledge to select models for model transformation testing. In: 1st Int. Conf. on Software Testing, Verification and Validation (ICST), pp. 328–337. IEEE (2008)
18. Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal specification and testing of model transformations. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) SFM 2012. LNCS, vol. 7320, pp. 399–437. Springer, Heidelberg (2012)

19. Wang, J., Kim, S.K., Carrington, D.: Automatic generation of test models for model transformations. In: 19th Australian Conf. on Software Engineering (ASWEC), pp. 432–440. IEEE (2008)
20. Weißleder, S., Sokenou, D.: Automatic test case generation from UML models and OCL expressions. In: Software Engineering 2008 - Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik. LNI, vol. 122, pp. 423–426. GI (2008)