https://doi.org/10.31449/inf.v47i1.4421 Informatica **47** (2023) 21–42 **21**

On Integrating Multiple Restriction Domains to Automatically Generate Test Cases of Model Transformations

Thi-Hanh Nguyen and Duc-Hanh Dang*
Department of Software Engineering,
VNU University of Engineering and Technology, Hanoi, Vietnam
E-mail: hanhit@hnue.edu.vn, hanhdd@vnu.edu.vn
*Corresponding author

Keywords: transformation testing, black-box testing, classifying term, model finding, OCL

Received: September 24, 2022

Testing model transformations poses several challenges, one of which is how to automatically generate effective test suites. A promising approach for this is to employ equivalence partitioning, a well-known technique for software testing. Specifically, in order to generate effective test suites, current works in literature often focus on exploiting either the structural aspects of models or transformation contracts for partition analysis. However, for the aim, they focus on only a single restriction source such as metamodels, contracts of the transformation, and domain-expert knowledge. To increase the effectiveness of generated test suites, partitioning techniques should be performed on a combination of various restriction sources. This paper introduces a method to generate test models on such a multi-domain of restrictions. The method also allows the tester to flexibly select and combine constraints to create a unified restriction for different strategies and objectives in model transformation testing. We developed a support tool based on the UML-based Specification Environment (USE) and performed experiments on several transformations to point out the effectiveness of our method.

Povzetek: Opisana je metoda preverjanja programske kode na osnovi multi-modalnih omejitev posameznih delov.

1 Introduction

Model transformations are the pillars of Model-Driven Engineering (MDE). Testing has been an effective technique to ensure the quality of model transformations which is the key to successfully realizing MDE in practice. This discipline consists of the following main tasks: synthesizing models as test data that are referred to as *test models*, performing the transformation, and verifying the output results. Until now, how to synthesize automatically and effectively *test models* for model transformations is still challenging.

The test model generation is the synthesis of models from different restriction sources including syntactic and semantic domains of source and target models. Such restriction domains often have complex structures and semantics that make it difficult to automate the generation. To the best of our knowledge, there are typical restriction domains in the context of MDE as follows. First, for a so-called *source metamodel coverage*, as explained in [1, 2, 3, 4, 5, 6], test models could be generated by applying the well-known testing technique *equivalence partitioning* that splits the input metamodel into equivalence partitions for selecting representative test models. Second, for a so-called *transformation specification coverage*, as proposed in [7, 8, 11], additional restrictions on source models could be derived from a transformation specification and taken as input contracts to generate test models. Within the works, input contracts of the transformation specification often are expressed as OCL conditions. Third, following the white-box testing approach, the works in [12, 13, 14, 15] focus on analyzing a model transformation implementation to build test suites using the notion of *transformation implementation coverage*. In addition, in interactive approaches, domain knowledge can support the test model selection. For example, based on the test objective, domain experts could choose representative values for the partition testing technique [1, 4, 16], or directly create examples for test models within test-driven development approaches, as explained in [18, 19].

Generating test models based on the analysis and synthesis of each single particular restriction domain can lead to a large duplication of test models, wasting testing time and effort. This highlights the need to generate test models from multiple restriction domains. However, realizing this need presents several challenges: (1) Constraints from multiple domains expressed in heterogeneous formalism need to be translated into a consistent and unified formalism to enable model synthesis. (2) The partition analysis technique is often employed to obtain representative test models since exhaustive testing is a non-trivial task, but defining a suitable partition on multiple restriction domains for different test strategies can be challenging. (3) The automatic generation of test models often requires manually defining (as input of

the solver) parameters for the testing environment as well as the other configuration information. This is challenging to automate this task.

This paper proposes a mechanism based on an integration of multiple restriction domains for a black-box testing approach to automatic generation of test models. Specifically, multi-domain restrictions that include (1) conditions for partitioning the metamodel and (2) transformation contracts are first translated into OCL conditions; and then taken as the input of a constraint solver for generating test models. For each common test strategy, a mechanism of combining OCL conditions should be established to define combinatorial partitions using logical operators. Moreover, a scope-value searching method needs to be incorporated to solve constraints and so that the set of generated test models has a reasonable size. The main contributions of this paper are summarized as follows:

– A method to automatically generate test models with multi-domain restrictions for effective model transformation testing.
– A mechanism to define suitable partitions for different test strategies.
– An OCL-based support tool and experimental results to show the effectiveness of the proposed method.

The rest of this paper is organized as follows. Section 2 surveys related works. Section 3 motivates this work with a transformation example. Section 4 outlines our approach. Section 5 explains restriction domains as a basis for a partition analysis and automatic generation of test models. Section 6 introduces several strategies to combine partitions in order to generate test models for different test objectives. Section 7 shows our tool support. Section 8 illustrates our testing method with several transformations and points out the effectiveness of our method. Section 9 explains threats to the validity of this work and discusses the results. This paper is closed with a conclusion and a discussion of future work.

2 Related work

In this section, we provide an overview of black-box testing approaches for model transformations and address the following research questions: (1) how to automatically generate test models using the partition analysis technique in a black-box testing approach, (2) how to construct test oracles that check test outputs to ensure quality properties; and (3) how to evaluate the quality of test suites in terms of one or more test objectives. First, a common basic idea of black-box testing approaches for transformations is to use metamodel and requirements specification as test basis, i.e., they are independent of transformation implementations. Within these approaches, the well-known testing technique *equivalence partition* [20] often is used to split the input data domain into equivalence partitions based on the test basis analysis and then to select a representative model for each partition. Fleurey et al. [1, 24] have proposed a partitioning technique based on the datatype of class attributes and the association end multiplicity within a UML class diagram representing the metamodel. Several other partitioning techniques for generating test models that conform to a metamodel as introduced in [2, 21, 4, 23, 5, 22, 6, 39, 9]. One of the main limitations of the metamodel-based partitioning approach is that the technique often generates a large number of test models, and generated test models tend to correspond to just only a subfragment of the source metamodel instead of the whole metamodel. To overcome this limitation, Fleurey introduces the notion of a so-called effective metamodel as the fragment of the source metamodel that is actually manipulated by the transformation. An effective metamodel can be defined by either examining the specification of transformations as explained in [1, 3] or statically analyzing their implementation as shown in [21].

The partitioning technique, as shown in [7, 10, 8, 9], can also be employed to specify requirements of model transformations. Following the research line, the works in [9, 39, 8] propose to derive partitioning conditions from a contract-based specification of the transformation. The specification of transformations within these approaches often includes preconditions and invariants as contracts on the input data domain of the transformation (i.e., corresponding to restrictions on source models). Partitioning conditions are then translated into either OCL constraints [7] or other first-order logic languages like Alloy [4, 23] for the automatic generation of test models using the model finding technique. For different test objectives, the works in [1, 6, 8, 4, 36] have proposed suitable techniques to improve the quality of test cases. Fleurey et al. [1] proposed the use of the Bacteriologic algorithm to optimize test suites. On analyzing metamodel-based partitioning, Fleurey et al. [1], Janabin et al. [6], Gogolla et al. [8], and Sen et al. [4] proposed using representative values provided by domain experts or testers. Similarly, Sen et al. [4] proposed combining different knowledge domains and uniformly representing them as constraints in Alloy. Cabot et al. [36] proposed a similar technique in which combinatorial partitioning conditions are represented in OCL.

Second, another major challenge for model transformation testing is how to predict desired expected outputs [1]. This research line can be divided into two groups: The first aims to predict the whole output model, i.e., making use of a *complete oracle function*, and the other aims to predict just part of desired target model, i.e., using a *partial oracle function*. The first approach (with complete oracle functions) would take the expected output model as a reference model and check if the actual output model conforms to this reference model, e.g., using model comparison as regarded in [32, 15, 17, 30, 34, 35]. For this aim, Addazi et al. [35] employs EMFCompare, whereas the works in [17, 15] design specific algorithms to compare models. Besides, Kolovos [33] employs the Epsilon Comparison

Table 1: Black-box approaches for test model generation (**MP**=> Metamodel-based Partitioning; **SP** => Specification-based Partitioning; **MF** => Model Finding; **Al** => Algorithm)

Reference	Test Model Generation			Restriction domains				Test adequacy	
	Partitioning	Representing patterns	Generating models	Metamodel	Specification	Transformation implementation	Domain Expert	Metamodel coverage	Specification coverage
Fleurey et al. [1]	MP	Pattern	AL	*				*	
Wang et al. [2, 21]	MP	Pattern		*		*		*	
Wu et al. [5, 22]	MP	OCL	AL	*			*	*	
Lamari [3]	MP, SP	Pattern		*		*		*	
Jahanbin et al. [6]	MP	Pattern	AL	*				*	*
Sen et al. [4, 23]	MP, SP	Alloy	MF	*	*		*		*
Guerra et al. [7]	SP	Pattern, OCL	MF	*	*				*
Burgueño et al. [9, 39]	MP	OCL	MF	*					*
Gogolla et al. [10, 27]	SP	OCL	MF	*	*				*

Language (ECL), a task-specific model management language, in order to define language-specific algorithms for model matching. The second approach (with partial oracle functions) aims to ensure certain properties of a transformation using the partial oracle functions. It is no need to manually define a whole expected target model within the approach. The works in [8, 28, 7] employ OCL contracts or OCL assertions as partial test oracles to express the expected properties of generated models and to automatically verify them. Contracts and assertions can be represented in the form of visual graph patterns as explained in [7, 31].

Testing is an informal approach for verifying the quality properties of model transformations. Depending on a given test objective, partial oracle functions aim to check whether the functional behavior of a transformation system fulfills such following properties: (1) *confluence*, *applicability* and *termination* which are called general properties; (2) *correctness* including syntactic and semantics correctness (for both information preservation and behavior preservation) and the completeness which are called specific properties. A specific property is often specific to a certain transformation specification. The analysis of general properties such as termination, determinism, rule independence, rule applicability, or reachability of system states requires to perform on a set of given transformation rules. This task is out of the scope of this paper.

The works in [28, 15, 7, 17, 1] propose to verify the syntactical correctness of transformations using test oracles captured from the target model's contracts. To check the source-target correspondence property [29], also known as the information preservation property, current approaches often employ source-target contracts represented either by OCL conditions [8, 7] or graph patterns [36] to consistently specify input test conditions and output test oracles. This work focuses on analyzing the impact of constraints used for test model generation on different transformation properties.

Test adequacy criteria measure the quality of a test suite regarding to several objectives. Test adequacy criteria help define testing goals to be achieved. In transformation testing, test adequacy criteria can be based on how well the test basis (e.g., the input metamodel and the transformation specification) is covered by the test models, or how effective the oracle functions are to identify synthetic bugs (so-called mutants) injected into the under-test transformation. As shown in the two last columns of Table 1, coverage-based approaches propose to measure the effectiveness of a black-box testing approach by evaluating how the input/output metamodels and/or the transformation specification are covered by the testing technique. Fleurey et al. [1] propose to measure the quality of a set of test models by measuring how much they cover the input metamodel. A measurement technique is defined in terms of class coverage, attribute coverage, and association coverage. The metamodel-coverage or effective metamodel-coverage are also introduced in several other works [2, 21, 5, 22, 6, 9, 39]. The notion of transformation specification coverage is introduced in [7, 8]. Within contract-based specifications, transformation contracts can be analyzed to define test conditions. For example, Guerra et al. [7] take preconditions and invariants as transformation properties and define test criteria that could cover these properties for generating test models. Test criteria could also be defined based on the combination of these properties within a combined testing strategy like t-way testing.

Additionally, mutation analysis approaches aim to measure the effectiveness of test cases based on their ability to detect bugs. Mottu et al. [50] propose exploring mutation analysis for model transformations. They study potential bugs that developers may bring into model transformations to define a set of generic mutation operators for model transformations. The mutation analysis technique is commonly used by current works in literature to effectively show the test case generated by proposed meth-

Table 2: Approaches for oracle function definition (**PO**=> Partial Oracle; **CO** => Complete Oracle; **SC** => Type Correctness/Syntactic Correctness; **IP** => Information Preservation)

Reference	Oracle type	Representing expected outputs	Automated	MT Properties
Guerra et al. [7]	PO	Pattern, OCL	*	SC, IP
Fleurey et al. [1]	PO	OCL	*	
Mottu el al. [31]	CO	Pattern		SC
Wieber et al. [15]	CO	Model		SC
Lano et al. [28]	PO	OCL	*	SC
Hilken et al. [8]	PO	OCL	*	SC
Lin et al. [17]	PO	Model	*	SC, IP
Troya et al. [30]	PO	Model		SC
Orejas et al. [34]	PO	Model		SC

ods [24, 7, 15].

3 Running example

This section motivates our work with the CD2RDBM model transformation between class diagrams (CD) and relational database models (RDBM). This transformation example is introduced in [46]. This paper focuses on its simplified version for common transformation situations as regarded in [25]. Metamodels specifying the input and output modeling spaces of the CD2RDBM transformations are shown in Fig. 1 and Fig. 2, respectively. Requirements of the CD2RDBM transformation contain constraints as restrictions on input/output models and the relationship between pairs of them. At the specification level, the requirements are independent of implementation language and often specified in the form of *transformation contracts*.

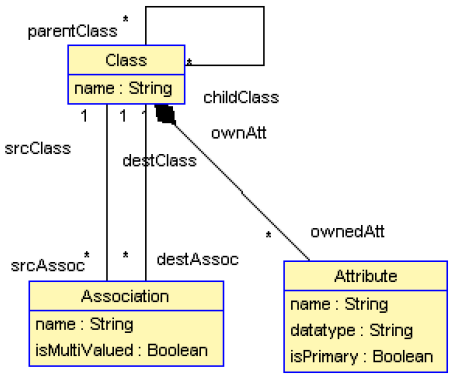

Figure 1: The simplified metamodel of class diagrams.

A transformation contract allows a designer to specify what a transformation does, under which conditions it can be applied to a model, and what its excepted result is. Such information is also helpful for choosing and applying the proper transformation in the context of off-the-shelf transformations. A contract-based model transformation specification typically consists of three sets of constraints corresponding to preconditions, postconditions, and invariants. First, *Preconditions* include constraints defining a set of models, each of which is a candidate as the source model. *Positive preconditions* state the expected properties on valid source models; *Negative preconditions* define source models that fulfill several forbidden properties, i.e., the source ones are invalid. For example, the CD2RDBM transformation includes the following precondition constraints:

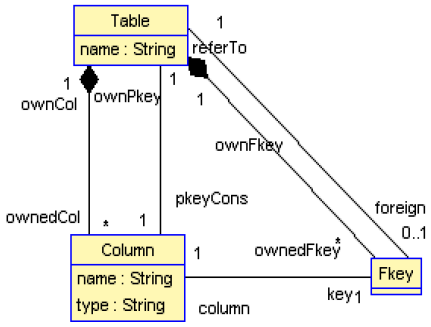

Figure 2: The simplified metamodel of relational database schema.

- A class does not inherit itself;
- The name of a class is unique;
- Attributes of a class must have distinctive names
- The child class does not redefine attributes of its parent class;
- The name of an association does not coincide with a class's name.

Second, *Postconditions* define a set of models produced by the transformation: *Positive postconditions* state the expected properties of valid target models; *Negative postconditions* define target models that satisfy several forbidden properties, i.e., the target ones are invalid. For example,

the CD2RDBM transformation includes the following post-conditions:

– A table name is unique;

– Two columns of a table must have distinct names;

– A table cannot have more than one primary key column.

Third, *Invariants* specify the correspondence between pair of source and target models, denoted by $ps \implies pt$. A positive (negative) invariant has the expressing structure: If the source model satisfies the property ps then the target model (does not) satisfies the property pt. As discussed in [26, 27, 7], the structure of each transformation rule also can be represented by a positive invariant that must hold between the source and target models to satisfy the transformation definition. The CD2RDBM transformation specification contains the following negative invariants:

– If the CD model has two classes with an inheritance relationship, the corresponding RDBM model could not have two distinction tables mapping to these classes.

– If the CD model has two mutually inherited classes, then the corresponding RDMB model could not only have the mapping table to the parent class while there is no mapping table to the child class.

– If the CD model has a class, the corresponding RDBM model could not have two distinction tables mapping to this class.

In addition, the CD2RDBM transformation has six mapping rules that define how a CD model is mapped to a corresponding RDBM model:

– A class must be mapped to a same-name table;

– The name and data type of a non-primary attribute coincides with the ones of a corresponding column;

– A primary attribute is mapped to a column played as the primary key;

– A multi-valued aggregation and association between two classes is mapped to a new associative table that relates the two corresponding tables;

– An aggregation/association relationship between two classes is characterized by a single-valued end and a multi-valued end (0..*, 1..*) is mapped to a foreign key that relates two corresponding tables;

– A child and its parent class are mapped to the same table.

Testing is required to find out if a model transformation is implemented and executed as expected for all possible inputs, or if there are bugs in the transformation leading to unintended output models for certain input models [29]. This model transformation can be realized using different transformation implementation languages. To test the quality of a model transformation captured from multiple restriction domains, a black-box testing approach is often employed.

Since exhausting testing is impossible, testing criteria are proposed to select representative test models to achieve the source metamodel coverage and the specification coverage.

Depending on the test objective, either the positive testing strategy or the negative testing strategy will be used to navigate the test case design and test execution process. The analysis of information on a test basis allows testers to determine test conditions in both negative testing and positive testing strategies.

4 Overview of the approach

Figure 3 overviews our approach to testing model transformations. The basic idea is to synthesize test models based on an integration of multiple restriction domains.

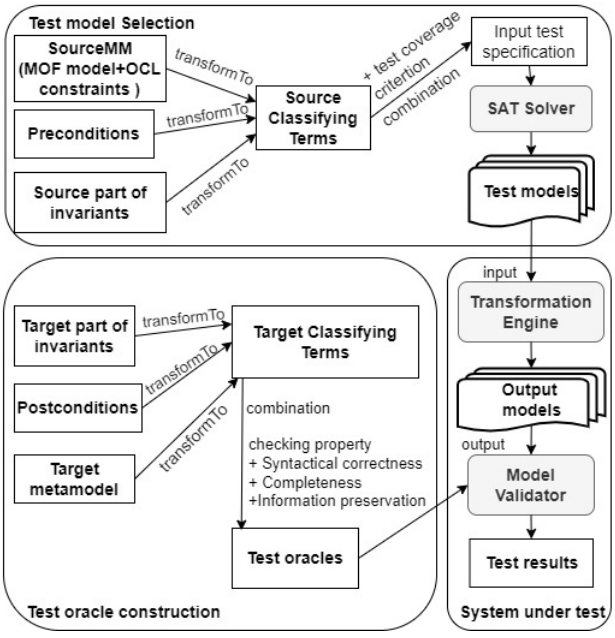

Figure 3: An integration of multiple restriction domains for model transformation testing.

First, the partitioning technique is employed to define test models that cover the source metamodel. The partitioning criteria that are either restrictions on the source metamodel or contracts of the transformation specification are expressed in the form of boolean OCL expressions, referred to as so-called classifying terms (CTs) [8]. In this way, the underlying conditions which are used in characterizing test models also can be flexibly combined to generate effective test suites.

Second, test criteria could be defined for both positive and negative testing strategies. To generate test models that

satisfy a test criterion, input test conditions captured from each restriction domain are expressed by classifying terms. The classifying terms are then combined and taken as the input of constraint solvers, including the SAT solver, in order to automatically generate test models.

Finally, to check test oracles, classifying terms derived from the target metamodel and the transformation specification are defined to ensure that (1) the output model conforms to the target metamodel, (2) the output model satisfies the postcondition, and (3) the output model must also comply with invariants that describe the transformation relationship between valid or invalid pairs of source and target models. Such test output evaluation conditions are then combined to evaluate expected properties on the output model using model validator tools, including OCL tools like USE.

5 Synthesizing test models from restriction domains

Test model selection involves finding valid and invalid input models within positive and negative testing strategies, respectively. Test models are generated by synthesizing models from different restriction sources. This section explains combining knowledge sources to generate valid and invalid models within the positive and negative test strategies based on the proposed covered criteria.

5.1 Metamodel coverage

In MDE, a metamodel is often represented in the form of a UML class diagram with the key meta-concepts of MOF [37] including classes, attributes, generalization, and associations. Therefore, test models, that conform to the source metamodel, can be defined by an equivalence partitioning on the class diagram [38] for the source metamodel with the two following criteria [1]:

- *AEM (Association End Multiplicities):* For each association end, each representative multiplicity must be covered. For instance, if an association end has the multiplicity [0..*], then it should be instantiated with the multiplicity 0, 1, and N (N is greater than 1).
- *CA (Class Attribute):* For each attribute, each representative value must be covered. For instance, representative values of a boolean attribute are $true$ and $false$ that define two corresponding partitions.

These criteria AEM and CA, as illustrated in Table 3, could be expressed in terms of representative values [1, 21, 4]. Representative multiplicity pairs can then be computed for an association by taking the Cartesian product of the possible multiplicities of each of its two ends. The representative values of each attribute can be computed from the typical data types of class attributes such as `Integer`, `String`, and `Boolean`.

Table 3: Representative values for multiplicities

Multiplicity property	Representative values
0..1	0, 1
1	1
0..*	0, 1, [>1]
1..*	1, 2, [>2]
N..*	N, N+1, [>(N+1)]
N..M	N, N+1, M-1, M

Boolean classifying terms (CTs) [39] are used to represent equivalence partitions for test models as follows. For each direction of an association between two classes, the name of the first class, the role name of the second class, and the multiplicity of the association ending at the second class are parameterized by variables $fClass$, $dClassRole$ and $sizeNumber$, respectively. Note that the $sizeNumber$ corresponds to the representative multiplicity value (as depicted in Table 3) at the second class. The parameter $fClass1$ is to define an arbitrary variant of instances of the first class. Using these parameters as input of the following OCL template, boolean CTs are generated from the metamodel.

```
fClass.allInstances -> exists( fClass1 |
  fClass1.dClassRole -> size() = sizeNumber )
```

Figure 4 shows a set of CTs for the simplified class diagram metamodel. Figure 5 demonstrates the partition analysis based on CTs captured from multiplicity values. Test suites with test models generated by the CT set would satisfy the association coverage.

This partitioning approach also includes the restriction on the data type of class attributes. Thus, generated test models could ensure the attribute coverage criterion [44, 2, 4, 3], i.e., each representative value of an attribute must be covered in at least one test model. The following example illustrates how representative values could be defined by analyzing the data range of primitive data types.

- The representative values for `Boolean` attributes are $\{true, false\}$;
- The representative values for `String` attributes: $\{null, '', ' something'\}$;
- The representative values for `Integer` attributes: $\{0, 1, > 1\}$.

The following OCL template is proposed to generate CTs for the attribute coverage criterion.

```
clsName.allInstances -> exists( varCls |
  varCls.attrName = rprValue )
```

In this OCL template, the parameter $attrName$ defines the attribute name, the $clsName$ defines the class name, the $rprValue$ defines the chosen representative value for the attribute data type, and the $varCls$ is to define an arbitrary variant of instances of the class. Figure 6 demonstrates the

```
1  Class.allInstances->exists(c1|c1.parentClass->size()=0)
2  Class.allInstances->exists(c1|c1.parentClass->size()=1)
3  Class.allInstances->exists(c1|c1.childClass->size()=0)
4  Class.allInstances->exists(c1|c1.childClass->size()=1)
5  Class.allInstances->exists(c1|c1.childClass->size()>1)
6  Class.allInstances->exists(c1|c1.ownedAtt->size()=0)
7  Class.allInstances->exists(c1|c1.ownedAtt->size()=1)
8  Class.allInstances->exists(c1|c1.ownedAtt->size()>1)
9  Attribute.allInstances->exists(a1|a1.owningAtt->size()=1)
10 Class.allInstances->exists(c1|c1.srcAssoc->size()=0)
11 Class.allInstances->exists(c1|c1.srcAssoc->size()=1)
12 Class.allInstances->exists(c1|c1.srcAssoc->size()>1)
13 Class.allInstances->exists(c1|c1.destAssoc->size()=0)
14 Class.allInstances->exists(c1|c1.destAssoc->size()=1)
15 Class.allInstances->exists(c1|c1.destAssoc->size()>1)
16 Association.allInstances->exists(a1|a1.srcClass->size()=1)
17 Association.allInstances->exists(a1|a1.destClass->size()=1)
```

Figure 4: CTs coverage representative values of association's multiplicities.

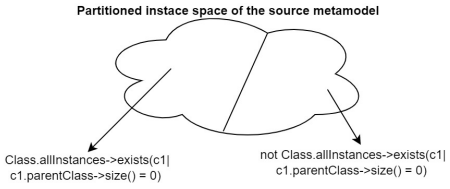

Figure 5: Partition analysis based on CTs captured from multiplicity values.

partition analysis based on CTs captured from the attribute's data type.

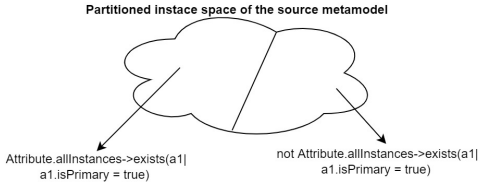

Figure 6: Partition analysis based on CTs captured from attribute's datatype.

There are two basic approaches to select representative values of equivalence partitions. The first, default partitioning chooses representative values using the boundary value analysis or the data random generation. The second, knowledge-based partitioning, representative values are provided by domain experts for various test objectives. This technique also allows the tester to flexibly adjust the configuration to narrow the searching space of constraint solving for a better test model generation.

Figure 7 shows a configuration file defined by the domain expert for the CD2RDBM transformation. Figure 8 shows classifying terms that partition source models based on the properties of the class *Class*.

5.2 Transformation specification coverage

A contract-based specification of a model transformation, brings benefits for debugging, testing, and, more generally, quality assurance. The partition analysis technique can be applied to contracts of a transformation specification to generate the test models that cover the transformation specification's requirements as regarded in [8, 36, 7]. This section explains a partition analysis technique based on classifier terms for model transformations. First, the underlying transformation will be captured by our TC4MT specification language [40]. The language TC4MT employs typed graph patterns in the form of a UML class added with OCL constraints to express transformation contracts. Transformation contracts can be either positive or negative.

Figure 9 shows a negative precondition specified in the TC4MT language. The precondition states that the CD2RDBM transformation rejects any input model in which a child class redefines an attribute of the parent class. Figure 10 shows a negative postcondition for the generated RDBM models. The example postcondition states a restriction on the output models that the column names of a table must be distinct.

Invariants within a transformation contract state how certain structures of an input model should be transformed. An invariant often consists of a source graph, a target graph, and an optional corresponding graph to connect them. A positive invariant that holds on a pair of source and target models would ensure there exists a target graph for each given source graph. With negative invariants, such a target graph should not be found from the target model domain. Figures 11 and 12 show a positive invariant and a negative invariant of the CD2RDBM transformation, respectively.

Considering each input condition on the input modeling space as a testing property, representative values of the property are defined for a testing partition. Then, graph patterns of representing input conditions are translated into boolean OCL expressions using the template as illustrated

```
------ Class'configuration information
Class = Set {c1,c2,c3,c4,c5,c6, c7,c8,c9, c10}
Class_name = Set{'','Person','Employee', 'Project', 'Department'}
Class_childClass = Set{0,1,2}
Class_parentClass = Set{0,1}
Class_ownedAtt = Set{0,1,2}
Class_srcAssoc = Set{0,1,2}
Class_destAssoc = Set{0,1,2}

------Attribute's configuration information
Attribute = Set {a1,a2,a3,a4,a5,a6, a7, a8, a9, a10}
Attribute_name = Set {'', 'eID', 'name', 'fullname', 'pID'}
Attribute_type = Set{'Integer', 'String', 'Boolean', 'Datetime'}
Attribute_isPrimary = Set{true, false}
Attribute_ownAtt = Set{1}

------Association's configuration information
Association = Set {ass1,ass2, ass3, ass4, ass5, ass6, ass7, ass8, ass9, ass10}
Association_name = Set{'', 'workFor', 'employee'}
Association_isMultiValued = Set{true, false}
Association_srcClass = Set{1}
Association_destClass = Set{1}
```

Figure 7: The configuration file for constructing source CTs.

in Fig. 13.

To translate graph patterns into OCL constraints, this schema will iterate over all objects of each contract (lines 2,3). In the case of negative contracts, i.e., the attribute $status$ of all objects equals to -2, the negation operator **not** appears at the first line of the schema. The function $conditions$ is to check a constraint on the underlying objects and their properties. If there exist two objects oi and oj with the same type ($type_{oi} = type_{oj}$) then the condition $oi <> oj$ will be added. The association between two objects will be translated into a corresponding condition, either $oi.role_j \rightarrow includes(oj)$ or $oj.role_i \rightarrow includes(oi)$. The condition function omits the checking of attributes with *undefined* value. Other OCL constraints of the graph pattern will be included in the function **conditions**. Figure 14 shows an OCL condition translated from the precondition shown in Fig. 9: *There does not exist any redefined attribute in the child class*.

A boolean OCL expression can be assigned to one of two values $\{true, false\}$ to specify a corresponding equivalence partition of the input model set. Models that violate a negative precondition will belong to an invalid equivalence partition of the input model set, while the other models will belong to the remaining partition of the input model set. Figure 15 illustrates the result of the partition analysis on preconditions.

Similarly, postcondition contracts as well as invariant contracts could also be translated into boolean OCL expressions to partition the output model set. These OCL expressions will be taken as OCL assertions playing the oracle function to verify actual output models.

6 Generating test models in different test strategies

Model transformation testing aims to ensure a transformation fulfills its requirements (i.e., validation testing) and to discover defects in the transformation (i.e., defect testing). For a certain test objective, the tester would follow a suitable test strategy. This section explains how test models are generated in such different test strategies.

Figure 16 depicts the workflow for test model generation. First, a test basic, including a transformation specification and a configuration of the test model domain, is analyzed and translated into boolean OCL expressions as classifying terms [8] to define partitioning information sets. Second, depending on different testing strategies, partitioning information sets and test criteria describe how the partitions are combined and selected to design test cases. Here, composite partitions are built according to certain specification coverage criteria. The test conditions in both test strategies are defined by combining single partitions using the relational operators $\{and, or, not\}$. Finally, these partition combinations are then taken as the input of an SAT solver [41] to automatically generate test models. For a particular OCL condition, the solver might not find any valid model since the given scope is too narrow, or there is inconsistency in the specification. In such cases, the search scope can be extended interactively by adjusting the solver parameters.

There are two main test strategies for model transformations: (1) A positive testing strategy aims to ensure correctness. This strategy focuses on generating valid input models. The tester could combine restriction domains cor-

```
Class.allInstances->exists(c1|c1.name='')
Class.allInstances->exists(c1|c1.name='Person')
Class.allInstances->exists(c1|c1.name='Employee')
Class.allInstances->exists(c1|c1.name='Project')
Class.allInstances->exists(c1|c1.name='Department')
Class.allInstances->exists(c1|c1.childClass->size()=0)
Class.allInstances->exists(c1|c1.childClass->size()=1)
Class.allInstances->exists(c1|c1.childClass->size()=2)
Class.allInstances->exists(c1|c1.parentClass->size()=0)
Class.allInstances->exists(c1|c1.parentClass->size()=1)
Class.allInstances->exists(c1|c1.ownedAtt->size()=0)
Class.allInstances->exists(c1|c1.ownedAtt->size()=1)
Class.allInstances->exists(c1|c1.ownedAtt->size()=2)
Class.allInstances->exists(c1|c1.srcAssoc->size()=0)
Class.allInstances->exists(c1|c1.srcAssoc->size()=1)
Class.allInstances->exists(c1|c1.srcAssoc->size()=2)
Class.allInstances->exists(c1|c1.destAssoc->size()=0)
Class.allInstances->exists(c1|c1.destAssoc->size()=1)
Class.allInstances->exists(c1|c1.destAssoc->size()=2)
```

Figure 8: Some source CTs generated from the partition analysis on the class $Class$.

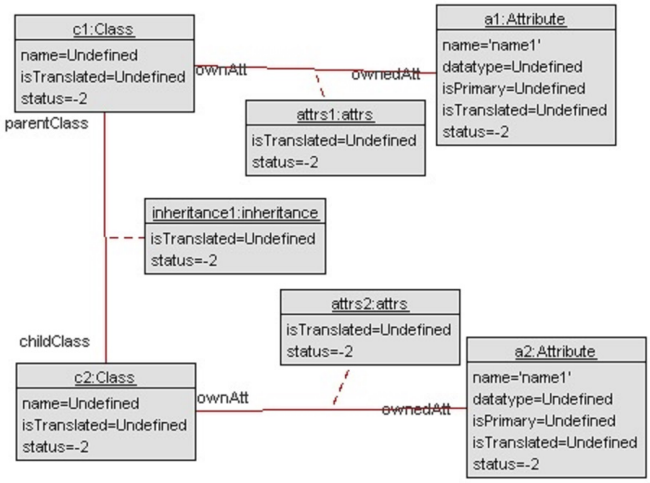

Figure 9: A negative precondition of the CD2RDBM transformation.

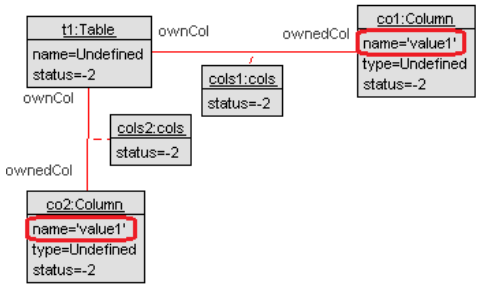

Figure 10: A negative postcondition of the CD2RDBM transformation.

responding to different aspects of the correctness property (including syntax correctness, semantic correctness, information preservation, and behavior preservation) to select relevant input models together with OCL assertions. (2) A negative testing strategy is applied to ensure safety and reliability. The strategy focuses on generating invalid input models so that transformation's defects might be detected.

6.1 Negative testing

Negative testing ensures that a model transformation can gracefully handle invalid input or unexpected execution scenarios. An input model is invalid if it violates at least one negative precondition. The equivalence partition technique is applied to preconditions of the transformation to identify various invalid partitions of input models.

To illustrate the negative testing approach, testers focus on the following typical situation. From a given negative precondition, two equivalence partitions are defined: a set of invalid input models that violate this precondition ($false$) and a set of the remaining models that fulfill this precondition ($true$). A model of the second set can be valid or invalid due to other remaining constraint conditions. Such a negative test case aims to discover defects when robustness testing. By combining many negative preconditions, a smaller partition of invalid input models would be defined.

To automate the generation of test inputs, a combination strategy is defined that describes how values ($true$ or $false$) for negative preconditions are selected such that the underlying coverage criterion is satisfied. The t-wise coverage criterion tends to be chosen for the negative testing approach. The coverage criterion is satisfied if any value combinations of t parameters, i.e., negative preconditions,

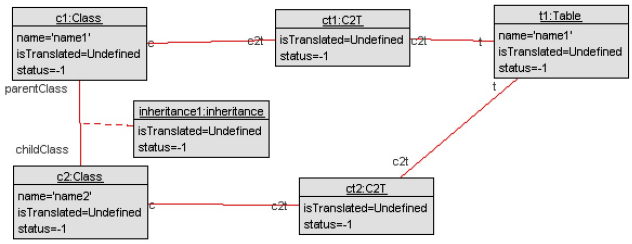

Figure 11: A positive invariant of the CD2RDBM transformation.

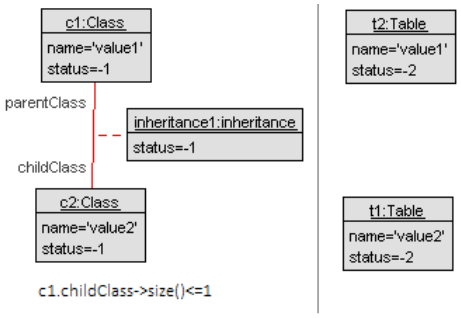

Figure 12: A negative invariant of the CD2RDBM transformation.

in this case, appear in at least one test input. As a special case of this, the following criteria are determined: each choice ($t = 1$), pair-wise ($t = 2$), and exhaustive ($t = n$).

Based on the combinatorial testing with negative test cases for software testing as explained in [42], different levels of specification coverage are defined for the negative test case generation as follows (see Fig. 17 for an illustration).

- **NP coverage:** For each negative precondition (the t-wise coverage with $t = 1$) at least one input model is selected.
- **2NP coverage:** For each negative precondition ($t = 1$) and each pair of negative preconditions ($t = 2$), at least one input model is selected.
- **Combinatorial NP coverage:** For each combination of t negative precondition ($t >= 1$), at least one test input model is selected. For instance, with the case $t = 4$, test input models would be generated for each negative precondition and each combination from 2 to 4 negative preconditions.

```
[not]
type_o1.allInstances->exists(o1|
    type_o2.allInstances->exists(o2|...
        type_on.allInstances->exists(on|
            conditions(o1,o2,...,on))))
```

Figure 13: The OCL schema for the precondition compilation.

```
not
Class.allInstances->exists(c1|
  Class.allInstances->exists(c2|
    Attribute.allInstances->exists(a1|
      Attribute.allInstances->exists(a2|
        c1<>c2 and
          a1<>a2 and
            c1.childClass->includes(c2) and
              c1.ownedAtt->includes(a1) and
                c2.ownedAtt->includes(a2) and
                  a1.name = a2.name))))
```

Figure 14: The OCL expression translated from the negative precondition shown in Figure 9.

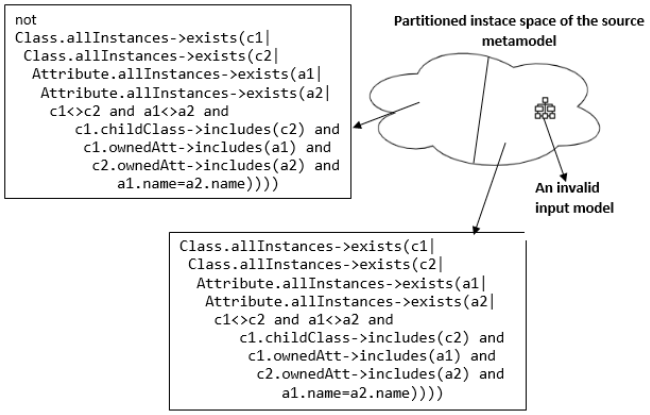

Figure 15: Partition analysis based on CTs captured from preconditions.

Figure 18 shows four test models generated by solving source classifying terms of the CD2RDBM transformation. These classifying terms are defined as a combination of negative preconditions. The first test model (M1) plays the role of a negative test case violating the negative precondition *NoSelfInheritance*. The remaining test models (M2,M3, and M4) are generated by the classifying term, defined by combining the two negative preconditions *NoSelfInheritance* and *NoDuplicateClassName*.

Linking negative test cases with test oracles. Negative testing ensures that a model transformation can gracefully handle invalid input data or unexpected user behavior. The purpose of negative testing is to prevent the system from crashing due to negative inputs and improve its quality and stability. The completeness property requires that the transformation refuses invalid input data and does not contain any incomplete execution. The syntactical correctness property requires that any output model produced from an invalid input model needs to be invalid, i.e., it violates at least one negative postcondition. The completeness of a transformation could be checked by performing negative

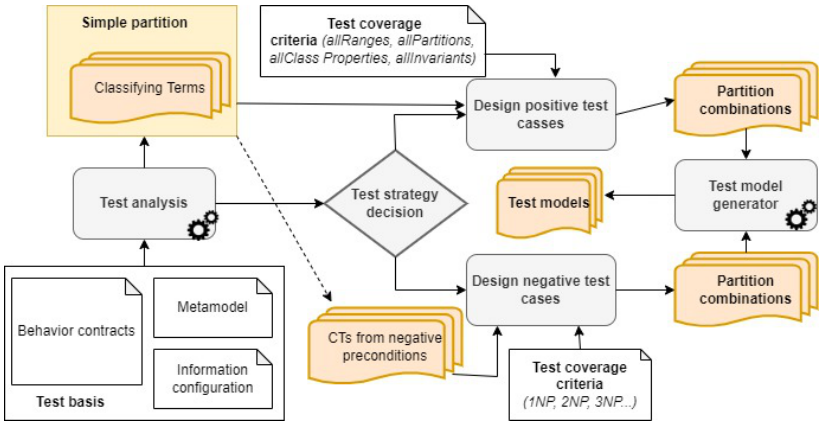

Figure 16: A test model generation process in different test strategies.

a) NP coverage

TC_id	P1	P2	P3
1	False		
2		False	
3			False

b) 2NP coverage

TC_id	P1	P2	P3
1	False	-	-
2	-	False	-
3	-	-	False
4	False	False	-
5	False	True	-
6	True	False	
7	False	-	False
8	False	-	True
9	True		False
10	-	False	True
11	-	True	False
12		False	False

c) Combinatorial coverage

TC_id	P1	P2	P3
1	False	-	-
2	-	False	-
3	-	-	False
4	False	False	-
5	False	True	-
6	True	False	
7	False	-	False
8	False	-	True
9	True		False
10	-	False	True
11	-	True	False
12		False	False
13	False	False	False
14	False	False	True
15	False	True	False
16	False	True	True
17	True	False	False
18	True	False	True
19	True	True	False

Figure 17: Partition analysis based on CTs captured from preconditions.

test cases and observing manually the execution process. The expected output of the execution is either an invalid input warning or a non-terminating state of the transformation system. Similarly, the syntactical correctness of the transformation also can be checked. The output model is now checked using the oracle function as shown in Fig. 19.

6.2 Positive testing

Positive testing verifies how the application behaves for the positive set of data. In positive transformation testing, single partitions are combined to select valid input models, representing composite partitions of the input model domain. Because valid input models must satisfy without violating any negative preconditions, all classifying terms translated from negative preconditions will be pushed into the SAT solver when solving constraints to generate valid input models.

A strategy to combine partition information in terms of classifying terms is defined to avoid duplication and reduce the number of test models. This strategy also ensures both the source metamodel coverage and the transformation specification coverage. The concept *Range* is denoted to a set of equivalent values of a class property and a*Partition* contains one and more *Range*. The following coverage criteria are proposed for the positive testing approach.

The *allRanges* coverage. The representative value of each range must be implemented in at least one test model. The following examples are model fragments of the metamodel(*MF*).

```
MF{Class.allInstances → exists(c|c.name='')}
MF{Class.allInstances → exists(c|c.name='var')}
MF{Class.allInstances → exists(c|c.childClass →
size()=0)}
```

The *allPartitions* coverage. The set of representative values of each *Partition* must appear in at least one test model. The following example model fragments are generated from this fragmentation criterion.

```
MF{Class.allInstances → exists(c|c.name='') ∧
Class.allInstances → exists(c|c.name='var')}
MF{Class.allInstances → exists(c| c.childClass
→ size()=0) ∧ Class.allInstances → exists(c|
c.childClass → size()=0) ∧ Class.allInstances →
exists(c| c.childClass → size()>1)}
```

A test model based on this coverage criterion can represent more constructs to be tested in the source metamodel than the *allRanges* coverage criterion. If an area is divided into three ranges, the tester can create a test model that corresponds to the three instances of the test model set in the *allRanges* criterion. Therefore, creating a suitable *allPartitions* coverage for a test model set can reduce the test case size while ensuring the metamodel coverage criterion.

The *allClassProperties* coverage. Each value association representing the partition of each class' attribute values must be implemented in at least one test model. The following example fragment models are generated from this

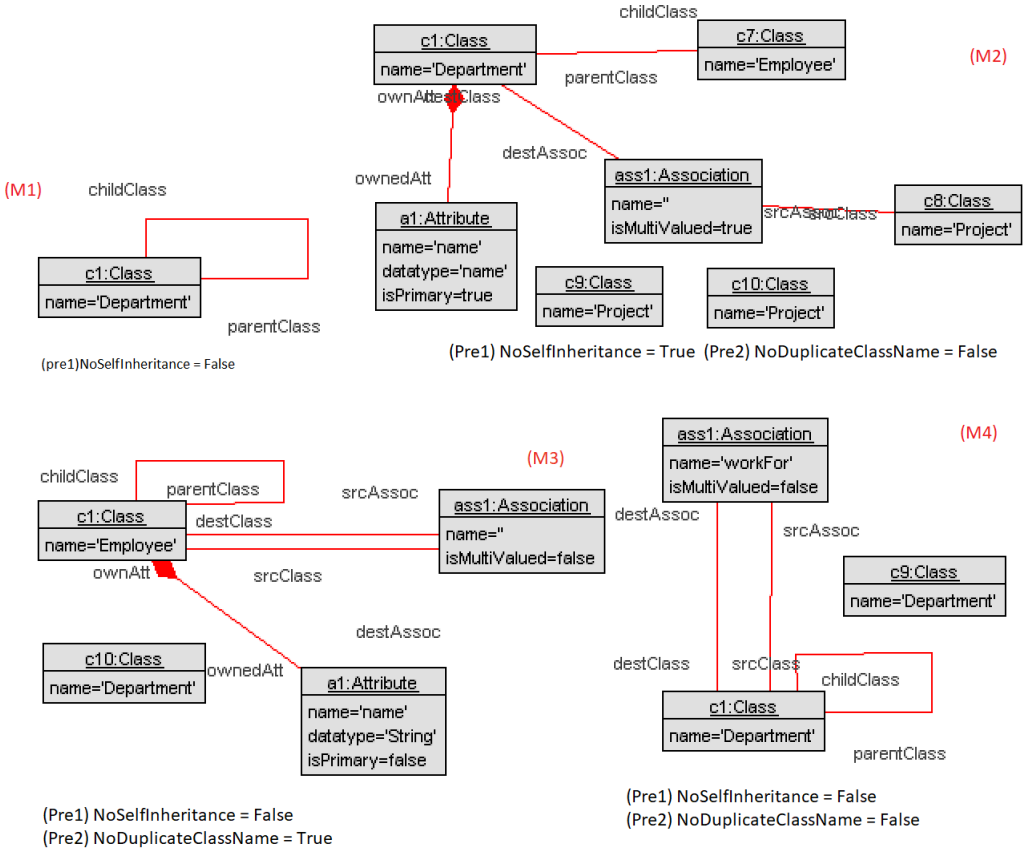

Figure 18: Four negative test cases generated from source CTs.

criterion.

```
MF{Class.allInstances → exists(c|c.name='var' ∧
c.childClass → size()=0 ∧ c.parentClass → size()=0
∧ c.ownedAtt → size()=0 ∧ c.srcAss → size()=0 ∧
c.destAss → size()=0)}
```

```
MF{Attribute.AllInstances → exists(a1| a1.name =
'attname') ∧ Attribute.AllInstances → exists (a1|
a1.datatype = 'atttype') ∧ Attribute.AllInstances →
exists(a1| a1.isPrimary = false)}
```

While the test coverage criteria *allRanges, allPartitions, allClassProperties* allow us to achieve the source metamodel coverage, specification-based test coverage criteria aim at the requirement coverage. A valid input model needs to fulfill all negative preconditions, therefore, test models generated by the positive testing strategy ensure the precondition coverage. In order to achieve the invariant coverage and to navigate the test input model selection, the following test coverage criterion is defined.

The *invariant* coverage. Each source pattern of invariants (consisting of both negative and positive invariants) needs to be implemented in at least one test model.

Positive w.r.t. negative invariants describe valid w.r.t. invalid pairs of source and target models. Therefore, the source graphs as part of an invariant can be used as templates, i.e., positive patterns, to generate test models in the positive testing strategy. The invariant coverage criterion requires that each source graph of invariants (including transformation rules) appears as a restriction on at least one test model. Considering the CD2RDBM transformation, with three negative invariants and six transformation rules, nine test models are required to satisfy the coverage invariant criterion.

Linking positive test cases with test oracles. The completeness property of a model transformation requires any valid input model also to be accepted as the input data and then transformed into an output model. In case all generated output models are valid target models satisfying all negative postconditions, the syntactical correctness property of a model transformation is ensured.

A model transformation is correct only if both input and output models are valid. In other words, the output model must preserve the information as well as the behavior of the input model through the transformation program. The correspondence between source (input) models and target (output) models can be captured by invariants. Therefore, invariants should be effective knowledge sources to check information preservation. Therefore, positive test cases including valid input models should be used as test data for checking the syntactical correctness and information preservation as shown in Fig. 20 and Fig. 21.

```
Function O1
  input:
    mtin: invalid input model
      mt: transformation implementation
  output:
    testResult: Boolean //true(pass), false(fail)
  do:
    mtout:=mt(mtin)
    if(mtout.satisfies(forAll(p)|p: negative postcondition))
      testResult:=false
    else
      testResult:=true
    endif
EndFunction
```

Figure 19: Oracle function for the syntactical correctness in the negative testing strategy.

```
Function O2
  input:
    mtin: valid input model
    mt: transformation implementation
  output:
    testResult: Boolean //true(pass), false (fail)
  do:
    mtout:=mt(mtin)
    if(mtout.satisfies(forAll(p)|p: negative postcondition))
      testResult:=true
    else
      testResult:=false
    endif
EndFunction
```

Figure 20: Oracle function for the syntactical correctness in the positive testing strategy.

```
Function O3
  input:
    inv{ps,pt}: positive invariant,
      ps is the source pattern of the inv,
      pt is the target pattern of the inv
    mtin: valid input model,
      mtin.satisfies(ps}
    mt: transformation implementation
  output:
    testResult: Boolean //true(pass), false (fail)
  do:
    mtout:=mt(mtin)
    if(mtout.satisfies(forAll(p)|p: negative postcondition))
      if(mtout.satisfies(pt}
        testResult:=true
      else
        testResult:=false
      endif
    else
      testResult:=false
    endif
EndFunction
```

Figure 21: Oracle function for the information preservation in the positive testing strategy.

7 Tool support

The USE (UML-based Specification Environment) [43] is the execution environment of the support tool of. The tool includes three main functional components as follows: (1) TC4MT specification tool; (2) Test generator; and (3) Test bench.

As shown in Fig. 22, the first component allows building the TC4MT transformation specification using the USE editor. In this component, the metamodel of a transformation specification is represented by a UML class diagram added with OCL constraints. Patterns of a specification are represented by object diagrams conforming to the metamodel. For example, the class diagram in Fig. 22 shows the metamodel of the CD2RDBM transformation specification while preconditions, postconditions, invariants, and transformation rules are represented by object diagrams created by using the graphical window interface or the scripting language SOIL of the USE editor.

The second component is a USE plugin that performs the specification analysis to define test conditions. Figure 23 shows the GUI of this component. The plugin is activated by loading a triple-type graph. The window *MetamodelAnalysis* (red label **1**) is used to automatically generate source classifying terms from the partition analysis on the source metamodel. An optional configuration file containing information provided by the domain expert can be loaded to increase the expressiveness of the source CTs. The window *SpecificationAnalysis* (red label **2**) is used to load patterns of preconditions, postconditions, and invariants (including transformation rules playing positive invariants) and translate them into CTs using the OCL schemes introduced in Section 5.2.

The last component, as shown in Fig. 24, is also a USE plugin playing the test bench. Test bench-related tasks are to use the Kodkod engine to solve OCL constraints for finding model instances playing test models, invoke the system under test (SUT) with the test models, and pass resulting output models to the oracle function for evaluation. This plugin is activated by loading the source metamodel. It takes as input the specification files of metamodels, transformation definition, source CTs, target CTs, and the Model Validator configuration, all of which are plain text files.

The transformation definition including a set of TGG-based rules is written in the RTL language [45] that can run on the USE tool. The configuration file (including value options for links, attributes, and size of elements) is required to restrict object models. The source CTs file is used to gen-

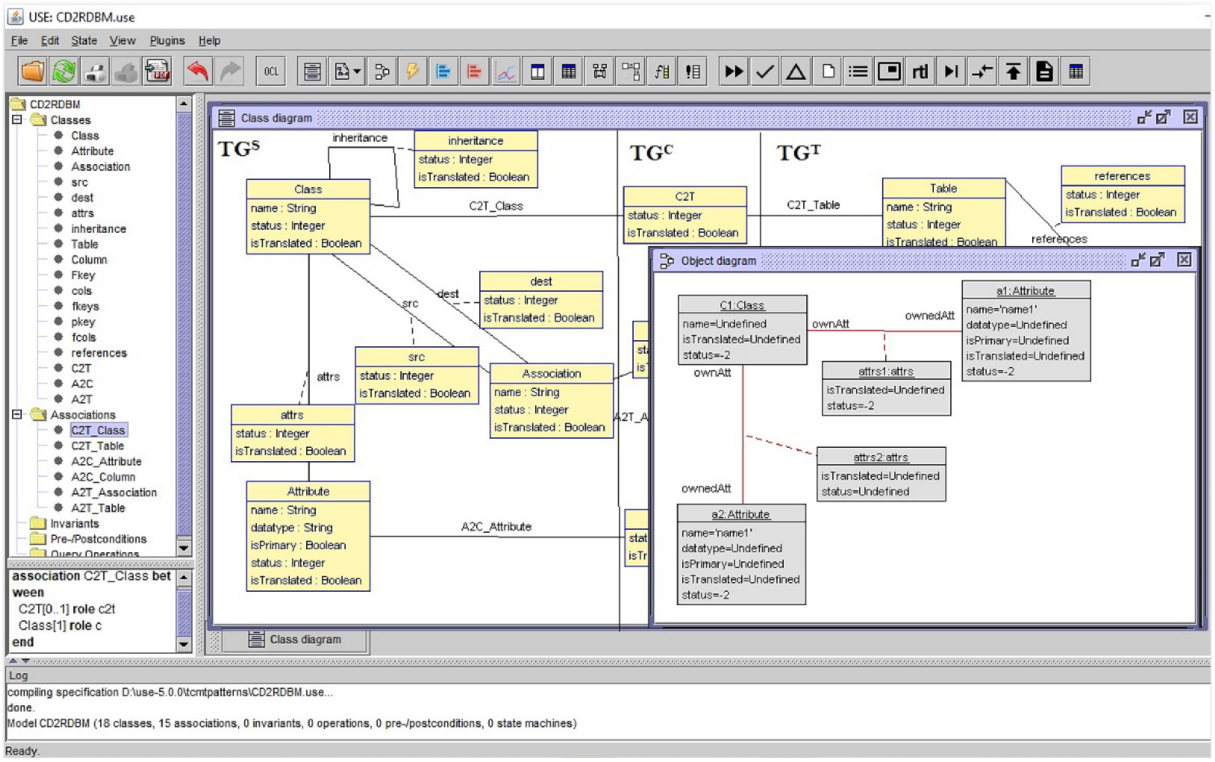

Figure 22: A transformation specification in the language TC4MT.

erate input models of test cases, the target CTs file is used to validate output models. The mapping file contains a list of patterns in the format of ”sourceCTs –> targetCTs”, in which each side specifies a list of expected Boolean values of CTs corresponding to passed test cases. The list of source CTs that are combined to represent the input test specification based on the selected test coverage criteria while corresponding target CTs are combined to represent the expected output property that defines the partial oracle function. The test report obtained from the test bench is shown in Fig. 25.

Finally, as shown in Fig. 25, when executing the test suite, each solution generated by SAT solver will be taken as input for the model transformation. The tool then reports whether the output model satisfies predefined OCL assertions. The partition information of each solution is presented in the panel *Source Classifying Terms*, as shown in Fig. 25. The validation result of the corresponding output model against OCL assertions is shown in the panel *Target Classifying Terms*. The oracle function that is predefined in the mapping file will check whether the test case is passed. The test result will be depicted in the panel *Validation result*. The transformation execution script is shown in the panel *Executed transformations*. The debugging of transformation execution scenarios is performed by invoking each rule application, step by step. The current state of the transformation system after each transformation step could be checked.

8 Experimental results

In this section, several experiments are performed to evaluate the effectiveness of generated input models for detecting transformation failures. The objective of the experiment is to evaluate the error detection ability of the designed test cases in both positive and negative testing strategies.

8.1 Tested setup

For the evaluation, the paper focuses on four model transformations written in the RTL implementation language, the Restricted graph Transformations Language, as proposed in [45]. The purpose of the transformation examples is as follows.

C2R. The CD2RDBM transformation [46] is implemented for the running example which includes six rules, five negative preconditions, three negative invariants, and three negative postconditions;

B2D. The BibTeX2DocBook transformation [47] transforms the BibTeX model into the XML-based format for document composition DocBook. However, in this paper, we are only interested in converting the information about proceedings of conferences presented in BibTeX models into corresponding information presented in DocBook models. The version of the transformation BibTeX2DocBook includes six rules, four

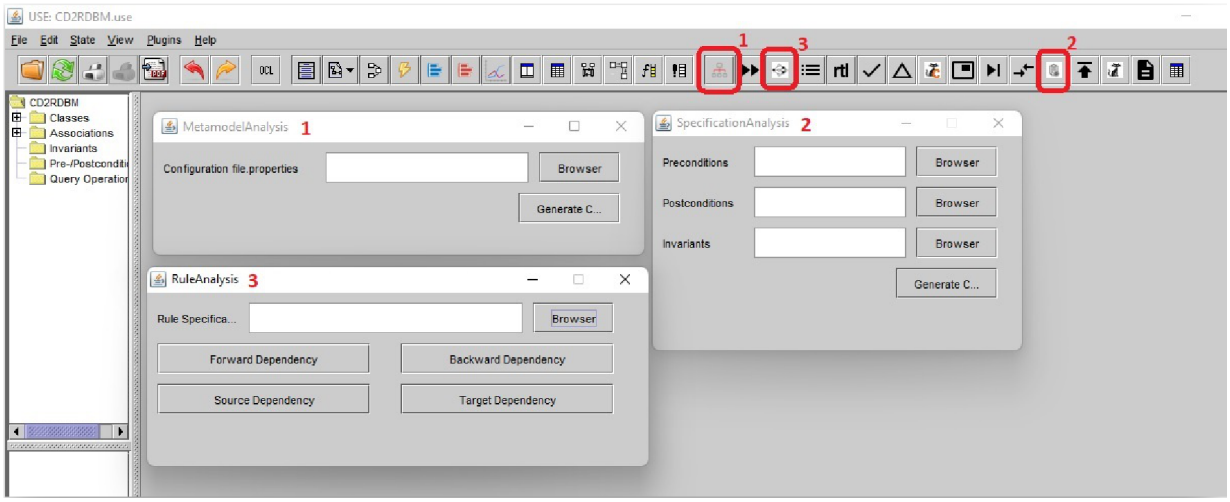

Figure 23: The GUI for the function to analyze transformation specifications.

negative preconditions, two negative invariants, and three negative postconditions;

F2P. The Families2Persons transformation [48] is part of the ATL transformation zoo and was created as part of the "Usine Logicielle" project. This transformation includes four rules, two negative preconditions, two negative postconditions, and no negative invariant;

B2P. The BPMN2PetriNet transformation [49] transforms BPMN models at the Computational Independent Model (CIM) level to PetriNet models at the Platform Independent Model (PIM) level. This model transformation includes twelve rules, five negative preconditions, two negative postconditions, and no negative invariant.

It is important to note here that the specification language TC4MT is independent of transformation implementation platforms. The transformation implementations need to conform to the transformation specification but are not derived from the TC4MT specifications automatically. Generated test suites can be used for verification and validation model transformation implementations using the black-box testing approach.

Table 4 gathers the number of contracts of each transformation example, as well as the size of its input metamodel. In our specification-based testing approach, we focus on negative preconditions, negative postconditions as well as negative invariants. Besides, transformation rules specify expected corresponding mappings between source models and target models so they can be considered as positive invariants of the source-target contracts.

The BPMN2PetriNet transformation is the most complex in terms of the size of specification as well as the input metamodel. The Families2Persons transformation is a simple transformation with few invariants, rules and classes.

8.2 Test suite generation

From the TC4MT specification, a test suite is derived based on each selected coverage criterion. The test suites were automatically generated by using the tool presented in Sect. 7. The numbers in side cell of Table 5 show the number of generated test models corresponding to a particular coverage criterion. In general, the larger the size of the specification is specified, the larger the test size is generated.

8.3 Efficacy of generated test suites

To measure the effectiveness of a test suite and help improve it, the common technique *mutation testing* [50] is employed. In mutation testing, faults are injected into a program to produce erroneous versions of it, which are called mutants. Then each mutant is tested with the test suite. Once the test suite could detect the error, the mutant is killed. Otherwise, the mutant remains alive. The mutant score, which is the number of killed mutants divided by the total number of mutants, gives a measure of the quality of the test suite.

The mutation testing technique is performed as follows. First, mutants of each transformation implementation are created manually by injecting faults by using systematic classification of mutation operators of model transformation regarded in [50].

Navigation. The model has navigated thanks to the relations defined on its metamodel and a set of elements is obtained. Therefore, navigation mutations replace the navigation towards a class with the navigation towards another, remove the last step of a chain of navigation, or add the last step of navigation in a navigation chain.

Filtering. A rule application is usually performed on a limited set of input and output model elements described by the filter conditions. Filtering mutations introduce

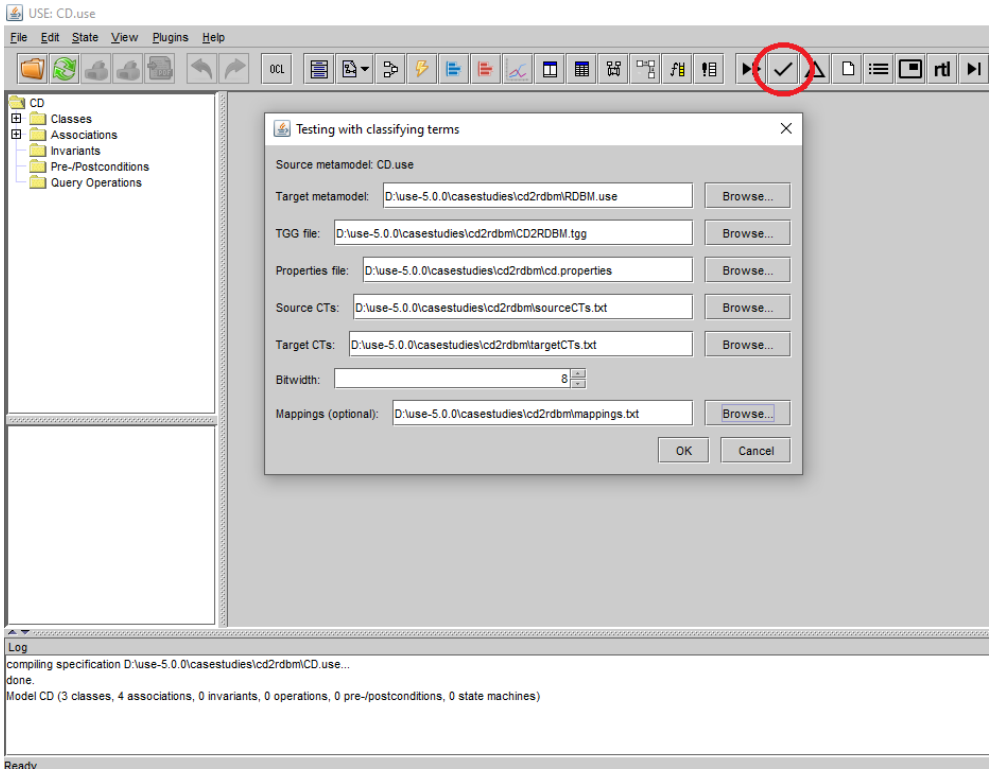

Figure 24: The GUI for the functions: test implementation, execution, and reporting.

Table 4: Test setup

Examples	Specification			Source Metamodel		
	Precond	Postcond	Inv+Rule	Class	Assoc	Inheritance
CD2RDBM	5	3	9	3	4	0
BibTeX2DocBook	4	3	8	5	3	2
Families2Persons	2	2	4	2	4	0
BPMN2PetriNet	5	2	12	15	2	14

disturbances in the filters of a collection, either by modifying the attributes used in the filter or by selecting only some instance types when the collection is defined with a generic class.

Creation. Output model elements are created by the execution of transformation rules. The creation mutations replace the creation of an object with another compatible type, delete the creation of a relation between two objects, or add a useless relation between two objects in the transformation rules of a transformation implementation.

Figure 26 shows an example of mutants. Here, the rule is specified in the RTL transformation language. The injected fault is highlighted in a colored square. In particular, the class $Column$ is related to the class $Table$ by two associations corresponding to the role names *ownPkey* and *ownCol*. This mutant aims to replace $c.ownPkey$ of the column c with the $c.ownCol$ so that the cardinality is modified.

Table 6 shows the mutation operators used to create the mutants in the experiment, which altogether belong to all possible mutation types (navigation, filtering, and creation). Each mutant was created by applying a mutation operator to the original transformation one time. Thus, each cell in the table corresponds to the number of mutants created using a particular mutation operator. The last column in the table summarizes the number of mutants created for each transformation.

Table 7 shows the number of mutants created from each transformation as well as the mutation score of the generated test suites using the negative testing strategy.

Table 8 shows the number of mutants created from each transformation using the positive testing approach.

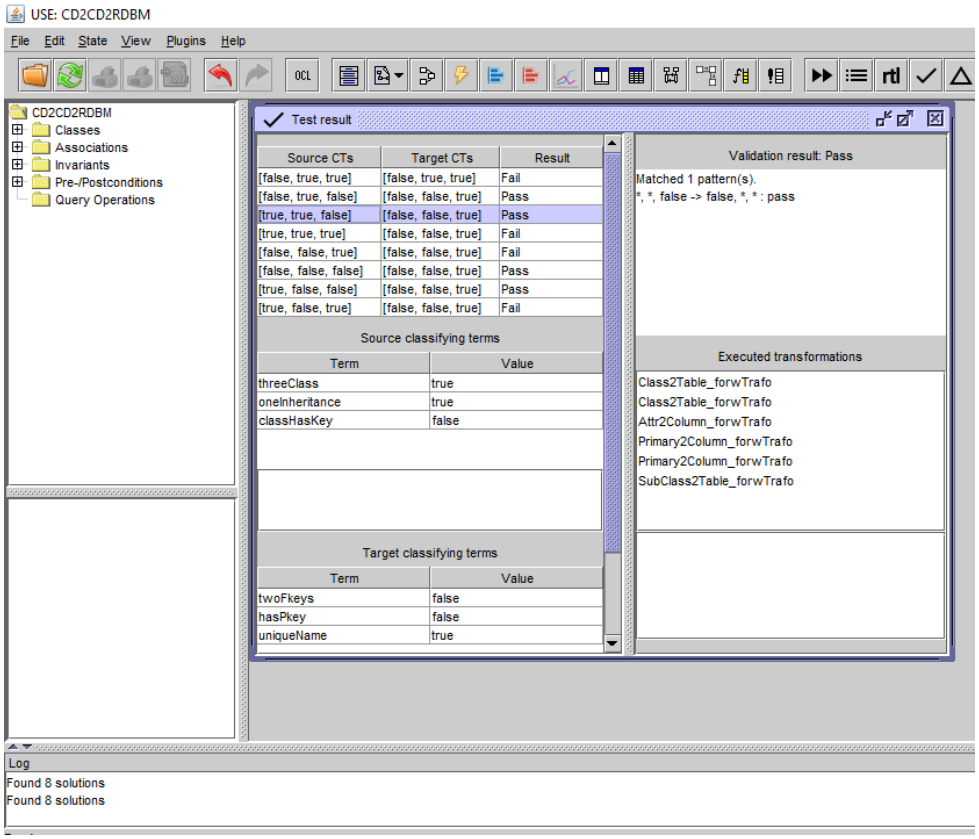

Figure 25: Test report with some partition information.

Table 5: The number of test models in test suites for coverage criteria

	Negative testing			**Positive testing**			
Examples	1NP	2NP	3NP	Ranges	Partitions	ClassProperties	Invariants
CD2RDBM	5	35	63	34	14	3	9
BibTeX2DocBook	4	22	50	24	12	5	8
Families2Persons	2	5	-	10	2	2	4
BPMN2PetriNet	5	35	63	53	30	15	12

Table 6: Number of mutants on the transformations CD2RDBM (C2R), BibTeX2DocBook (B2D), Families2Persons (F2P), and BPMN2PetriNet (B2P)

	Navigation	Filtering	Creation	Total
C2R	9	28	21	58
B2D	6	12	7	25
F2P	12	16	4	32
B2P	13	36	20	69

Table 7: Mutation scores of the generated test suites in the negative testing strategy

	Mutants	1NP	2NP	3NP
C2R	58	0.90	0.90	0.90
B2D	25	0.84	0.84	0.88
F2P	32	0.81	0.81	-
B2P	69	0.62	0.62	0.81

9 Threats to Validity

Although we performed the experiments with utmost care, some underlying parameters potentially threaten the validity of the obtained results:

i) We experimented with common transformation examples that are available in related works. However, we only specify and implement these transformation examples with simplified requirements and particular fragments of input/output metamodels in-

Table 8: Mutation scores of the generated test suites in the positive testing strategy

	Mutants	allRanges	allPartitions	allClassProperties	allInvariants
CD2RDBM	58	0.90	0.90	0.90	0.90
BibTeX2DocBook	25	0.72	0.80	0.80	0.72
Families2Persons	32	0.75	0.75	0.75	0.75
BPMN2PetriNet	69	0.65	0.70	0.70	0.65

```
---r3(n:String,t:String)---single-value-primary-attribute--------
rule PrimaryAtt2Column
checkSource(
    C: Class
    [not Attribute.allInstances->forAll(a|a.isPrimary=true
       and a.owner->includes(C))]
){
    A: Attribute
    (C, A): attrs
    [A.isPrimary = true]
    [A.name<>Undefined]
    [A.datatype<>Undefined]
}checkTarget(
    T: Table
    --original statement
    ---[not Column.allInstances->exists(col|col.ownPkey->includes(T))]
    --mutant with injected fault
    [not Column.allInstances->exists(col|col.ownCol->includes(T))]
){
    Co: Column
    (T, Co): cols
    (T, Co): pkey
}checkCorr(
    (C,T) as (c,t) in c2t:C2T
){
    (A,Co) as (a,c) in a2c:A2C
    A2C:[self.c.name = self.a.name]
    [self.c.type = self.a.datatype]
}end
```

Figure 26: An example for the mutation operator *navigation*.

stead of whole metamodels. For example, in the BibTeX2DocBook transformation, we only work on BibTeX files representing the conference proceedings. Mutation scores of generated test suites generally are dependent on specific factors such as the way to create mutants, the size of test suites, and the quality of the under-test transformation implementation. Therefore, the obtained experimental results only point out the error-detection efficiency of generated test suites for typical semantic faults of transformations. Our mutation-based evaluation method is inapplicable for the other specific faults.

ii) We empirically evaluate transformation examples realized by the RTL bidirectional transformation language designed based on the integration of TGG and OCL [45]. Because of the flexibility of the OCL language, there can be different implementations for the transformation from a specification. Therefore, the number of created mutants for each different implementation does not coincide with each other. Note that the RTL implementation currently is not derived automatically from the TC4MT specification although it also has the TGG-based semantics. Even in the case of the automatically generated implementation, testing such an implementation would affect the evaluation results. This makes RTL implementations more objective assessment.

iii) In the workflow of our proposed approach, several steps are still performed manually and interactively, such as the step to create configuration files containing representative values of partitions, the one to create mutant sets, and the one to select the solver configuration. Therefore, the quality of the tester's work and their decisions should have more or less an impact on the experimental results.

Discussion. As surveyed in Sect. 2, current black-box testing approaches often employ meta-model coverage criteria to ensure that the set of generated input models contains at least one instance of any class or association of the meta-model. They also refer to extreme values for the attributes. However, a limitation of the approaches is that a very large number of test models, including unrelated or duplicated test models, are generated, and the completely generated test model is often not related to the test output evaluation. Several testing approaches focus on contract-based model transformation specification analysis to generate smaller test model sets using the specification-based coverage criteria. An advantage of these approaches is that the test models remain intentional: They are generated for testing a particular combination of transformation requirements so that they can be checked by the oracle function more efficiently.

In this paper, a testing approach is proposed that combines different knowledge sources to generate smaller, more efficient test model suites with different test objectives. This combination reduces test model duplication while still ensuring efficient metamodel coverage and specification coverage. In our approach, the use of environment configuration parameters provided by domain knowledge makes generated test suites more efficient. Test models are designed by using both negative testing and positive testing strategies. The approach allows us to verify further quality properties of a model transformation. Some test oracle functions also are defined for verifying quality properties against appropriate test suites generated by different testing strategies. To show the effectiveness of the generated test cases in detecting common semantic errors, some experiments are performed on different transformation examples as regarded in Sect. 8.

10 Conclusion and future work

This paper proposes a specification-driven testing approach for test data selection. The basic idea is to leverage different sources of knowledge that can be produced during the transformation specification development and to utilize them for automatic generation of test suites. Different sources of knowledge as restriction domains are translated into OCL conditions to facilitate the partitioning testing. The experiments show that boolean OCL expressions could be combined to synthesize test models. Based on the characteristics of knowledge sources and selected testing strategies, input model conditions would be linked with output model assertions to check different quality properties.

The proposal testing framework, named TC4MT, employs SAT solver for finding test models automatically. The TC4MT framework is installed to support automated testing of RTL transformation implementation on USE environments. Several experiments are conducted in which test suites are automatically generated from several transformation specifications. We then measured the efficacy of the generated tests using the mutation analysis. The quality of the generated test set highly relies on how complete a specification is. If a specification only covers part of the transformation requirements, then the generated models may not enable the testing of the underspecified parts.

The performance and scope of test model searching remain a challenge for the proposed approach, we plan to conduct further experiments to improve performance and test coverage.

Acknowledgements

This work has been supported by Vietnam National University, Hanoi under Project No. QG.20.54. We wish to thank the anonymous reviewers for numerous insightful feedback on the first version of this paper.

References

[1] F. Fleurey, J. Steel, and B. Baudry (2004). Validation in Model-Driven Engineering: Testing Model Transformation. *First International Workshop on Model, Design and Validation*, IEEE, pp. 29-40, https://doi.org/10.1109/modeva.2004.1425846

[2] J. Wang, S.K. Kim, and D. Carrington (2006). Verifying metamodel coverage of model transformations. *In Proc. of Australian Software Engineering Conference*, IEEE, pp. 270–282, https://doi.org/10.1109/aswec.2006.55

[3] M. Lamari (2007). Towards an automated test generation for the verification of model transformations. *ACM Symposium on Applied Computing (SAC)*, ACM, pp. 998–1005, https://doi.org/10.1145/1244002.1244220

[4] S. Sen, B. Baudry, and J.M. Mottu (2008). On combining multi-formalism knowledge to select models for model transformation testing. *Software Testing, Verification and Validation (ICST)*, IEEE, pp. 328–337, https://doi.org/10.1109/icst.2008.62

[5] H. Wu, R. Monahan, and J. F. Power (2013). Exploiting Attributed Type Graphs to Generate Metamodel Instances Using an SMT Solver. *In Proc. of TASE*, pp. 175–182, https://doi.org/10.1109/tase.2013.31

[6] S. Jahanbin and B. Zamani (2018). Test Model Generation Using Equivalence Partitioning. *In Proc. of ICCKE*, pp. 98–103, https://doi.org/10.1109/iccke.2018.8566335

[7] E. Guerra (2012). Specification-driven test generation for model transformations. *In Proc. of ICMT* , pp. 40–55, https://doi.org/10.1007/978-3-642-30476-7_3

[8] F. Hilken, M. Gogolla, L. Burgueño, and A. Vallecillo (2018). Testing models and model transformations using classifying terms. *Software and Systems Modeling*, pp. 885-912, https://doi.org/10.1007/s10270-016-0568-3

[9] L. Burgueño, J. Cabot, R. Clarisó, and M. Gogolla (2019). A Systematic Approach to Generate Diverse Instantiations for Conceptual Schemas. *In Proc. of ER*, pp. 513–521, https://doi.org/10.1007/978-3-030-33223-5_42

[10] M. Gogolla, J. Bohling, and M. Richters (2005). Validating UML and OCL models in USE by automatic snapshot generation. *Software and Systems Modeling*, pp. 386–398, https://doi.org/10.1007/s10270-005-0089-y

[11] Stephan Hildebrandt, Leen Lambers, Holger Giese (2013). Complete Specification Coverage in Automatically Generated Conformance Test Cases for TGG Implementations. *In Proc. of ICMT* , pp. 174-188, https://doi.org/10.1007/978-3-642-38883-5_16

[12] J.M. Küster and M. Abd-El-Razik (2006). Validation of Model Transformations - First Experiences Using a White Box Approach. *Models in Software Engineering, Workshops and Symposia at MoDELS 2006*, pp. 193-204, https://doi.org/10.1007/978-3-540-69489-2_24

[13] C.A. Gonzalez and J. Cabot (2012). Atltest: A white-box test generation approach for ATL transformations. *In Proc. of Model Driven Engineering Languages and Systems*, pp. 449–464, https://doi.org/10.1007/978-3-642-33666-9_29

[14] D. Calegari and A. Delgado (2013). Rule Chains Coverage for Testing QVT-Relations Transformations. *In Proc. of the Second Workshop on the Analysis of Model Transformations (AMT 2013)*, pp. 449-464.

[15] M. Wieber, A. Anjorin, and A. Schürr (2014). On the Usage of TGGs for Automated Model Transformation Testing. *Theory and Practice of Model Transformations*, pp. 1-16, https://doi.org/10.1007/978-3-319-08789-4_1

[16] B. Alkhazi, C. Abid, M. Kessentini, D. Leroy, and M. Wimmer (2020). Multi-criteria test cases selection for model transformations. *Autom. Softw. Eng*, 27(1): pp. 91-118, https://doi.org/10.1007/s10515-020-00271-w

[17] Y. Lin, J. Zhang, and J. Gray (2005). A testing framework for model transformations. *In Model Driven Software Development*, pp. 219–236, https://doi.org/10.1007/3-540-28554-7_10

[18] L. Lengyel and H. Charaf (2015). Test-driven verification/validation of model transformations. *Frontiers of Information Technology and Electronic Engineering*, pp. 85-97, https://doi.org/10.1631/fitee.1400111

[19] J.S. Cuadrado (2020). Towards Interactive, Test-driven Development of Model Transformations. *Journal of Object Technology*, 19(1), pp. 1-22, https://doi.org/10.5381/jot.2020.19.3.a18

[20] T. J. Ostrand, M. J. Balcer (1988). The category-partition method for specifying and generating functional tests. *Communications of the ACM*, pp. 676-686, https://doi.org/10.1145/62959.62964

[21] J. Wang, S.K. Kim, and D. Carrington (2008). Automatic generation of test models for model transformations. *In Proc. of Australian Conf. on Software Engineering*, IEEE, pp. 432–440, https://doi.org/10.1109/aswec.2008.4483232

[22] H. Wu (2016). Generating metamodel instances satisfying coverage criteria via SMT solving. *In Proc. of MODELSWARD*, pp. 40–51, https://doi.org/10.5220/0005650000400051

[23] S. Sen, B. Baudry, and J.M. Mottu (2009). Automatic Model Generation Strategies for Model Transformation Testing. *In Proc. of ICMT*, pp. 148–164, https://doi.org/10.1007/978-3-642-02408-5_11

[24] F. Fleurey, B. Baudry, P. Muller, and Y. Le Traon (2009). Qualifying input test data for model transformations. *Software and Systems Modeling*, pp. 185–203, https://doi.org/10.1007/s10270-007-0074-8

[25] H. Ehrig, C. Ermel, U. Golas, and F. Hermann (2015). Graph and Model Transformation - General Framework and Applications. *Monographs in Theoretical Computer Science*, Springer, pp. 5-399, https://doi.org/10.1007/978-3-662-47980-3

[26] J. Cabot, R. Clarisó, E. Guerra, J.D. Lara (2010). Verification and validation of declarative model-to-model transformations through invariants. *J. Syst. Softw*. 83(2), pp. 283-302, https://doi.org/10.1016/j.jss.2009.08.012

[27] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and Lars Hamann (2012). Formal Specification and Testing of Model Transformations. *Formal Methods for Model-Driven Engineering*, Springer, pp. 399-437, https://doi.org/10.1007/978-3-642-30982-3_11

[28] K. Lano, S. Fang, and S.K. Rahimi (2020). Model Transformation Specification and Verification. *In Proc. of International Conference on Quality Software*, pp. 45-54, https://doi.org/10.1002/9780470522622.ch14

[29] A.R. Lukman and J. Whittle (2013). A survey of approaches for verifying model transformations. *Software and Systems Modeling*, pp. 1003-1028, https://doi.org/10.1007/s10270-013-0358-0

[30] J. Troya, S. Segura, and A. Ruiz-Cortés (2018). Automated inference of likely metamorphic relations for model transformations. *Journal of Systems and Software (2018)*, pp. 188–208, https://doi.org/10.1016/j.jss.2017.05.043

[31] M. Mottu, S. Sen, M. Tisi, and J. Cabot (2012). Static Analysis of Model Transformations for Effective Test Generation. *In Proc. of ISSRE*, pp. 291–300, https://doi.org/10.1109/issre.2012.7

[32] S. Mazanek and C. Rutetzki (2011). On the importance of model comparison tools for the automatic evaluation of the correctness of model transformations. *In Proc. of IWMCP*, pp. 12–15, https://doi.org/10.1145/2000410.2000413

[33] D. S. Kolovos (2009). Establishing Correspondences between Models with the Epsilon Comparison Language. *In Proc. of ECMDA-FA*, pp. 146–157, https://doi.org/10.1007/978-3-642-02674-4_11

[34] F. Orejas and M. Wirsing (2009). On the specification and verification of model transformations. *In: Palsberg, Semantics and Algebraic Specification*, vol. 5700 of Lecture Notes in Computer Science, pp. 140–161, https://doi.org/10.1007/978-3-642-04164-8_8

[35] L. Addazi, A. Cicchetti, J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio (2016). Semantic-based Model Matching with EMFCompare. *In Proc. of ME*, pp. 40–49.

[36] A.C. Carlos and J. Cabot (2014). Test Data Generation for Model Transformations Combining Partition and Constraint Analysis. *In Proc. of International Conference on Model Transformation*, pp. 25-41, https://doi.org/10.1007/978-3-319-08789-4_3

[37] http://www.omg.org/spec/MOF/

[38] A.A. Andrews, R.B. France, S. Ghosh, and G. Craig (2003). Test adequacy criteria for UML design models. *Softw. Test. Verification Reliab*, 13(2), pp. 95-127, https://doi.org/10.1002/stvr.270

[39] L. Burgueño, F. Hilken, A. Vallecillo, and M. Gogolla (2016). Generating effective test suites for model transformations using classifying terms. *In Proc. of PAME/VOLT*, pp. 48–57, https://doi.org/10.1007/s10270-016-0568-3

[40] T.H. Nguyen and D.H. Dang (2021). A Graph Analysis Based Approach for Specification-Driven Testing of Model Transformations. *NAFOSTED Conference on Information and Computer Science*, pp. 224-229, https://doi.org/10.1109/nics54270.2021.9701514

[41] E. Torlak, and D. Jackson (2007). Kodkod: A Relational Model Finder. *In Proc. of International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pp. 632-647, https://doi.org/10.1007/978-3-540-71209-1_49

[42] K. Fögen, H. Lichter (2019). Combinatorial Robustness Testing with Negative Test Cases. *In Proc. of International Conference on Software Quality, Reliability and Security*, pp. 34-45, https://doi.org/10.1109/qrs.2019.00018

[43] M. Gogolla, F. Büttner, and M. Richters (2007). USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1), pp. 27-34, https://doi.org/10.1016/j.scico.2007.01.013

[44] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and L.T. Yves (2006). Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. *Symposium on Software Reliability Engineering*, IEEE, pp. 85-94, https://doi.org/10.1109/issre.2006.27

[45] D.H. Dang (2009). On integrating triple graph grammars and OCL for model-driven development. *University of Bremen, Ph.D. thesis*, 2009, https://doi.org/10.1007/978-3-642-01648-6_14

[46] https://www.eclipse.org/atl/atlTransformations/

[47] A.G. Domínguez and G. Hinkel (2019). The TTC 2019 Live Case: BibTeX to DocBook. *In Proc. of the 12th Transformation Tool Contest, co-located with the 2019 Software Technologies: Applications and Foundation*, pp. 61-65.

[48] A. Anjorin, T. Buchmann, and B. Westfechtel (2017). The Families to Persons Case. *In Proc. of the 10th Transformation Tool Contest (TTC 2017)*, pp. 27-34.

[49] Z. Li, X. Zhou, Z. Ye (2019). A Formalization Model Transformation Approach on Workflow Automatic Execution from CIM Level to PIM Level. *International Journal of Software Engineering and Knowledge Engineering*, pp. 1179-1217, https://doi.org/10.1142/s0218194019500372

[50] J.M. Mottu, B. Baudry, and Y. L. Traon (2006). Mutation analysis testing for model transformations. *In Proc. of Model Driven Architecture - Foundations and Applications*, 2nd European Conference, pp. 376–390, https://doi.org/10.1007/11787044_28