Software and System Modeling manuscript No.
(will be inserted by the editor)

Automatic Test Model Generation to Transformation Pre-condition Refinement

A Symbiotic Methodology To Validate Model Transformations

Sagar Sen, Jean-Marie Mottu, Juan Cadavid, Benoit Baudry

ATLANMOD, Ecole des Mines, 44000 Nantes, France
e-mail: sagar.sen@mines-nantes.fr
Université de Nantes (IUT de Nantes / LINA - UMR CNRS 6241),
3 rue du Maréchal Joffre - 44000 Nantes
e-mail: jean-marie.mottu@univ-nantes.fr
INRIA Rennes - Bretagne Atlantique / IRISA, Université Rennes 1,
Triskell Team, Campus de Beaulieu, 35042 Rennes Cedex, France
e-mail: {jcadavid, bbaudry}@irisa.fr

Received: date / Revised version: date

Abstract Testing a *model transformation* requires input test models which effectively cover the input domain of the transformation. In order to reduce testing costs and increase error-revealing power of test models, it is necessary to automate the generation of these models using systematic criteria.This automation faces two key challenges: (1) accurate specification of the transformation's input domain (2) automatic generation of test models in the input domain based on effective coverage criteria. This paper presents a global approach that addresses these challenges. Typically the input domain defines a possibly infinite set of models specified using various sources of knowledge: the input metamodel of the transformation, static semantic constraints on this metamodel and pre-conditions that further constrain the input domain for a particular transformation. We use our tool PRAMANA to automatically generate a finite number of test models in the input domain using coverage criteria based on partitioning properties of the input metamodel. Testing the transformation with these models often reveals that some test models are not executable by the transformation although they satisfy the constraints of the input domain specification. This gives way to incremental refinement of the input domain specification using newly constructed pre-conditions until all generated test models are executable by the transformation. In our experiments, we validate the proposed approach with mutation analysis to empirically evaluate error-revealing power of the test models we generate using PRAMANA. The empirical evaluation uses the representative transformation of simplified UML class diagram models to RDBMS models. The evaluation is based on 3200 automatically generated test models. We demonstrate that partitioning strategies gives mutation scores of up to 93% vs. 72% in the case of unguided generation.

1 Introduction

Model transformations are core *Model Driven Engineering* (MDE) components that automate important steps in software development such as refinement of an input model, refactoring to improve maintainability or readability of the input model, aspect weaving into models, exogenous/endogenous transformations of models, and the classical generation of code from models. Although there is wide spread development of model transformations in academia and industry. However, there is mild progress in techniques to test transformations [1]. In this paper, we address the problem of testing model transformations using automatically generated test models. Our approach is applicable to a diverse set of transformation languages such as those based on graph rewriting [2], imperative execution (Kermeta [3]), and rule-based transformation (ATL [4]).

Testing a model transformation requires a *set of test models* in the input domain of a model transformation. The automatic synthesis/generation of such test models that can reveal bugs in a transformation is the subject of this paper.

The automatic generation of test models selects test models from the set of all input models. This set of input models is precisely specified by the input domain of a model transformation. Typically, this input domain specification relies on knowledge from various sources: (1) the input metamodel of the transformation (2) invariants/constraints on the static semantics of the input metamodel (3) pre-condition contracts for a particular transformation. We call the cumulative set of these sources of knowledge the *input domain model*. The input domain model specifies potentially an infinite set of input models. Therefore, we need to go a step further and define effective strategies, such as coverage criteria [5], that can help automatically select a finite number of test models in the infinite set.

In this paper, we build on previous work [6] to automatically generate test models using the tool PRAMANA (previously known as CARTIER) within the input domain of a transformation. Executing the model transformation with these test models often results in some test models being rejected by the transformation. Although these test models conform to the initial specification of the input domain model, they are rejected by the transformation. This may happen when the transformation runs into an infinite cyclic loop while navigating an input model. At this point there are two possibilities: (a) Modifying a model transformation's specification and consequently its implementation to handle such an input model (b) Creating a pre-condition that avoids input models with a certain pattern such as a loop. This is a dilemma about whether to correct a model transformation to make it robust or improve the pre-condition to handle unforeseen inputs. The choice of one approach or the other depends on how we would like to interprets the specification. In this paper, we consider taking step (b) to identify the *true input domain* of a model transformation conforming to a specification. Our *first contribution* in the paper is as follows:

Incremental Pre-condition Improvement: We automatically generate test models using PRAMANA to first identify test models outside the unknown *true* input domain of a model transformation. We systematically compose new pre-conditions using information extracted from a malicious pattern in the generated test models. We improve the current set of pre-conditions of the model transformation. We use the new specification of the input domain model to generate test models and improve the set of pre-conditions. We continue the process until no new pre-condition is required and all generated test models can be executed by the transformation. The output of the process is the precise model of the true input domain of a model transformation. Pre-condition improvement is an approach adapted to black-box transformations where we have no access to the implementation. It also reveals the unforeseen requirements in case we want to evolve a model transformation. Therefore, a pre-condition may be converted to a new model transformation feature. We see pre-condition improvement as either a means to protect the model transformation or to extract requirements in the form of constraints/rules for the evolution of the transformation.

Once, we have a precise input domain model the consequent next step is to generate new test models. This time to detect bugs in a transformation. This brings us to the *second contribution* of the paper:

Automatic Generation and Evaluation of Test Models: We first use PRAMANA to generate sets of test models for different strategies in the precise input domain of a model transformation. In this paper, we generate test models that satisfy coverage criteria [5]. Second, we use *mutation analysis* [7] [8] for model transformations as technique to evaluate if these test sets can indeed reveal bugs. Mutation analysis consists of artificially injecting *model transformation specific* bugs into a model transformation giving a set of mutant model transformations with exactly one bug in each mutant. We execute each of these mutants with a set of generated test models for a given strategy. A difference in output between the original transformation and a mutant transformation reveals that an error was detected for the same input test model. Consequently, the number of such errors detected by a set of test model refers to the *mutation score* of the test set. This number of errors detected corresponds to the *mutation score* of a test set which is the metric we use to compare test generation strategies. Mutation analysis is done mainly to evaluate the effectiveness of our coverage criteria to generate test models that detect bugs. However, mutation analysis is not used in practice for testing. We use mutation analysis to gain confidence that test models generated with PRAMANA can detect bugs for the representative transformation. Based on our mutation analysis results we encourage generation of test models using our coverage criteria in new scenarios involving arbitrary input domains and transformations.

We demonstrate incremental pre-condition improvement and empirically evaluate automatic test model generation for the representative model transformation of Unified Modelling Language Class Diagram (UMLCD) to Relational Database Management Systems (RDBMS) models called class2rdbms. We discover nine new pre-conditions for the class2rdbms. Using mutation analysis we demonstrate that our input domain coverage strategies, previously presented in [5], can select test models with considerably higher bug detection abilities (93%) compared to unguided selection (72%) in the input domain.. These results are based on 3200 generated test models and several hours of computation on a 10 machine grid of high-end servers. The large difference in mutation scores between coverage strategies and unguided selection can be attributed to the fact that coverage strategies enforce several aspects on test models that unguided selection fails to do. For instance, coverage strategies enforce injection of *inheritance* in the UMLCD test models. Unguided strategies do not enforce such a requirement. Several mutants are killed due to test models containing inheritance.

The paper is organized as follows. In Section 2 we present the transformation testing problem and the running case study. In Section 3, we present foundational ideas used in PRAMANA. In Section 4, we describe the PRAMANA methodology for incremental pre-condition improvement and the empirical approach for automatic test model generation. In Section 5, we present the experimental setup for test model generation using different strategies and discuss the results of mutation analysis. In Section 6 we present related work. We conclude in Section 7.

2 Problem Description

We present the problem of testing *model transformations*. A model transformation $MT(I,O)$ is a program applied on a set of input models I to produce a set of output models O as illustrated in Figure 1. The set of all input models is specified by a metamodel MM_I (For example, simplified UMLCD in Figure 2). The set of all output models is specified by metamodel MM_O. The pre-condition of the model transfor-

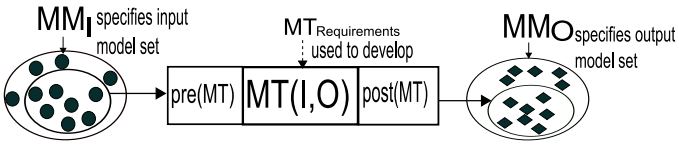

Fig. 1 A Model Transformation

mation $pre(MT)$ further constrains the input domain. A post-condition $post(MT)$ constrains the model transformation to producing a subset of all possible output models. The model transformation is developed based on a set of textual specification of requirements $MT_{Requirements}$.

Automatic model generation for testing involves finding valid input models we call *test models* from the set of all input models I. Test models must satisfy constraints that increase the trust in the quality of these models as test data and thus should increase their capabilities to detect bugs in the model transformation $MT(I,O)$. Bugs may also exist in the input metamodel and its invariants MM_I or the transformation pre-condition $pre(MT)$. However, in this paper we only focus on generating input models that can detect bugs in a transformation. In the process, we also generate input models that cannot be processed by a transformation (as they were unforeseen in $MT_{Requirements}$) which are then used to specify new pre-conditions.

2.1 Transformation Case Study

Our case study is the transformation from simplified UML Class Diagram models to RDBMS models called class2rdbms. In this section we briefly describe class2rdbms and discuss why it is a representative transformation to validate test model generation strategies.

In testing we need input models that conform to the input metamodel MM_I and transformation pre-condition $pre(MT)$. Therefore, we only discuss the MM_I and $pre(MT)$ for class2rdbms and avoid discussion of the model transformation output domain. In Figure 2, we present the simplified UMLCD input metamodel for class2rdbms. The concepts and relationships in the input metamodel are stored as an Ecore model [9] (Figure 2 (a)). The invariants on the simplified UMLCD Ecore model, expressed in Object Constraint Language (OCL) [10], are shown in Figure 2 (b). The Ecore model and the invariants together represent the input domain for class2rdbms. The OCL and Ecore are industry standards used to develop metamodels and specify different invariants on them. OCL is not a domain-specific language to specify invariants. However, it is designed to formally encode natural language requirements specifications independent of its domain.

The input metamodel MM_I gives an initial specification of the input domain. However, the model transformation itself has a pre-condition $pre(MT)$ that test models need to satisfy to be correctly processed. Constraints in the pre-condition for class2rdbms include: (a) All Class objects must have at least one primary Property object (b) The type of an Property object can be a Class C, but finally the transitive closure of the type of Property objects of Class C must end with type PrimitiveDataType. In our case we approximate this recursive closure constraint by stating that Property object can be of type Class up to a depth of 3 and the 4th time it should have a type PrimitiveDataType. This is a finitization operation to avoid navigation in an infinite loop. (c) A Class object cannot have an Association and an Property object of the same name (d) There are no cycles between non-persistent Class objects.

We choose class2rdbms as our representative case study to validate input selection strategies. It serves as a sufficient case study for several reasons. The transformation is the benchmark proposed in the MTIP workshop at the MoDELS 2005 conference [11] to experiment and validate model transformation language features. The input domain metamodel of simplified UMLCD covers all major metamodelling concepts such as inheritance, composition, finite and infinite multiplicities. The constraints on the simplified UMLCD metamodel contain both first-order and higher-order constraints. There also exists a constraint to test transitive closure properties on the input model such as there must be no cyclic inheritance. The class2rdbms exercises most major model transformation operators such as navigation, creation, and filtering (described in more detail in [8]) enabling us to test essential model transformation features. Among the limitations the simplified UMLCD metamodel does not contain Integer and Float attributes. There are also no inter-metamodel references and arbitrary containments in the simple metamodel.

Model generation is relatively fast but performing mutation analysis is extremely time consuming. Therefore, we perform mutation analysis on class2rdbms to qualify transformation and metamodel independent strategies for model synthesis. If these strategies prove to be useful in the case of class2rdbms then we recommend the use of these strategies to guide model synthesis in the input domain of other model transformations as an initial test generation step. For instance, in our experiments, we see that generation of a 15 class simplified UMLCD models takes about 20 seconds and mutation analysis of a set of 20 such models takes about 3 hours on a multi-core high-end server. Generating thousands of models for different transformations takes about 0.25% of the time while performing mutation analysis takes most of the time.

3 Foundations

This section presents foundational ideas used by the methodology for automatic test model generation and empirical evaluation presented in Section 4. First, we present the modelling and model transformation language Kermeta in Section 3.1. We use Kermeta to implement all model transformations in the paper. We briefly describe PRAMANA for automatic test model generation in Section 3.2. Effective test model generation in this paper is guided by coverage criteria based testing strategies. These testing strategies are described in Section 3.3. Finally, the bug detecting effectiveness of test models generated using different testing strategies is done by muta-

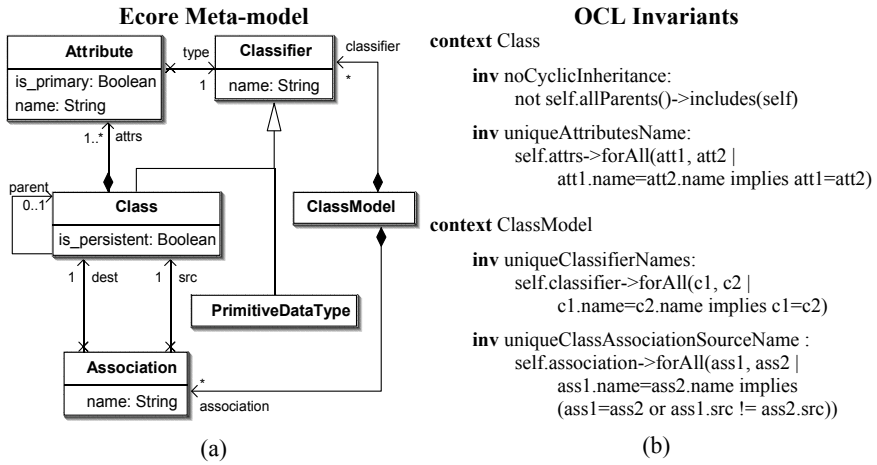

Fig. 2 (a) Class Diagram Subset of UML Ecore Meta-model (b) OCL constraints on the Ecore metamodel

tion analysis. Mutation analysis for model transformations is described in Section 3.4.

3.1 Kermeta

Kermeta is a language for specifying metamodels, models, and model transformations that are compliant to the Meta Object Facility (MOF) standard [12]. The object-oriented metalanguage MOF supports the definition of metamodels in terms of object-oriented structures (packages, classes, properties, and operations). It also provides model-specific constructions such as containments and associations between classes. Kermeta extends the MOF with an *imperative action language* for specifying constraints and operational semantics for metamodels [13]. Kermeta is built on top of EMF within the ECLIPSE development environment. The action language of Kermeta provides mechanisms for dynamic binding, reflection, and exception handling. It also includes classical control structures such as blocks, conditionals, and loops.

3.2 PRAMANA*: A Tool for Automatic Model Generation*

We use the tool PRAMANA previously introduced (with the name CARTIER) in our paper [6] to automatically generate test models. PRAMANA transforms the input domain specification of a model transformation to a common constraint language ALLOY. Solving the ALLOY model gives zero or more models in the input domain of a transformation. PRAMANA first transforms a model transformation's input metamodel expressed in the Eclipse Modelling Framework [9] format called Ecore using the transformation rules presented in [6] to ALLOY. Basically, classes in the input metamodel are transformed to ALLOY signatures and implicit constraints such as inheritance, opposite properties, and multiplicity constraints are transformed to ALLOY facts.

Second, PRAMANA also addresses the issue of transforming invariants and pre-conditions on metamodels expressed in the industry standard Object Constraint Language (OCL) to ALLOY. The automatic transformation of OCL to ALLOY presents a number of challenges that are discussed in [14]. We do not claim that all OCL constraints can be manually/automatically transformed to ALLOY for our approach to be applicable in the most general case. The reason being that OCL and ALLOY were designed with different goals. OCL is used mainly to query a model and check if certain invariants are satisfied. ALLOY facts and predicates on the other hand enforce constraints on a model. This is in contrast with the side-effect free OCL. The core of ALLOY is declarative and is based on first-order relational logic with quantifiers while OCL includes higher-order logic and has imperative constructs to call operations and messages making some parts of OCL more expressive. In our case study, we have been successful in transforming all meta-constraints on the UMLCD metamodel to ALLOY from their original OCL specifications. Nevertheless, we are aware of OCL's status as a current industrial standard and thus provide an automatic mapping to complement our approach.

Previous work exists in mapping OCL to ALLOY. The tool UML2Alloy [15] takes as input UML class models with OCL constraints. The authors present a set of mappings between OCL collection operations and their ALLOY equivalents. Here we present our version of such transformation derived from [15] and written in Kermeta.

The context of an OCL constraint (which is what defines the value of the *self* constraint) determines the place of the constraint within the generated ALLOY model. It is added as an appended fact. The mappings in Table 1 (taken in part from [15]) show the automatic transformation rules applied in PRAMANA.

However, some classes of OCL invariants cannot be automatically transformed to ALLOY using the simple rules in Table 1. For example, consider the invariant for no cyclic inheritance in Figure 2(b) [16]. The constraint is specified as the fact in Listing 1. This is an example in which the richness of the ALLOY language overcomes OCL - it is not possible to

specify this constraint in OCL without using recursive queries since there is no transitive closure operator.

```
fact noCyclicInheritance
{
  no c: Class | c in c.^parent
}
```

Listing 1 ALLOY Fact for No Cyclic Inheritance

The generated ALLOY model for the UMLCD metamodel using PRAMANA is given in Appendix A. This ALLOY model describes the input domain of the transformation.

3.3 Test Selection Strategies

Effective strategies to guide automatic model generation are required to select test models that detect bugs in a model transformation. We define a strategy as a process that generates ALLOY *predicates* which are constraints added to the ALLOY model synthesized by PRAMANA as described in Section 4. This combined ALLOY model is solved and the solutions are transformed to model instances of the input metamodel that satisfy the predicate. We present the following strategies to guide model generation:

Table 1 Mappings from OCL to ALLOY

Let *v* be a variable, *col* a collection, *expr* an expression, *be* an expression that returns a boolean value, *o* an expression that returns an object, *T* a type, *propertyCallExpr* an expression invoking a property on an object

OCL Expression Type	ALLOY Abstract Syntax Type
$context\ T\ inv\ expr$	$sig\ T\{\ldots\}\{expr\}$
$col \rightarrow forAll(v:T \mid be)$	$all\ v:T \mid be$
$col \rightarrow forAll(v:col \mid be)$	$all\ v:col \mid be$
$expr1 and expr2$	$expr1\ \&\&\ expr2$
$expr1 or expr2$	$expr1 \mid\mid expr2$
$not\ be$	$!be$
$col \rightarrow size()$	$\#col$
$col \rightarrow includes(o:T)$	$o\ in\ col$
$col \rightarrow excludes(o:T)$	$o\ !in\ col$
$col1 \rightarrow includesAll(col2)$	$col2\ in\ col1$
$col1 \rightarrow excludesAll(col2)$	$col2\ !in\ col1$
$col \rightarrow including(o:T)$	$col\ +\ o$
$col \rightarrow excluding(o:T)$	$col\ -\ o$
$col \rightarrow isEmpty()$	$no\ col$
$col \rightarrow notEmpty()$	$some\ col$
$expr.propertyCallExpr$	$expr.propertyCallExpr$
$if\ be\ then\ expr1\ else\ expr2$	$be \Rightarrow expr1\ else\ expr2$
$expr.oclIsUndefined$	$\#expr = 0$
$expr \rightarrow oclIsKindOf(o:T)$	$expr\ in\ o$
$col1 \rightarrow union(col2)$	$col1 + col2$
$col1 \rightarrow intersection(col2)$	$col1\ \&\ col2$
$col1 \rightarrow product(col2)$	$col1 \rightarrow col2$
$col \rightarrow sum()$	$sum\ col$
$col1 \rightarrow symmetricDifference(col2)$	$(col1 + col2) - (col1 \& col2)$
$col \rightarrow select(be)$	$v:col \mid be$
$col \rightarrow isUnique(propertyCallExpr)$	$no\ disj\ v1, v2 : col \mid v1.propertyCallExpr = v2.propertyCallExpr$

– **Random/Unguided Strategy:** The basic form of model generation is unguided where only the ALLOY model obtained from the metamodel and transformation is used to generate models. No extra knowledge is supplied to the solver in order to generate models. The strategy yields an empty ALLOY predicate as shown in Listing 2.

```
pred random { }
```

Listing 2 Empty ALLOY Predicate

– **Input-domain Partition based Strategies:** We guide generation of models using test criteria to combine *partitions* on domains of all properties of a metamodel (cardinality of references or domain of primitive types for attributes). A *partition* of a set of elements is a collection of *n* ranges A_1,..., A_n such that A_1, ..., A_n do not overlap and the union of all subsets forms the initial set. These subsets are called *ranges*. We use partitions of the input domain since the number of models in the domain are infinitely many. Using partitions of the properties of a metamodel we define two coverage criteria that are based on different strategies for combining partitions of properties. Each criterion defines a set of *model fragments* for an input metamodel. These fragments are transformed to predicates on metamodel properties by PRAMANA. For a set of test models to cover the input domain at least one model in the set must cover each of these model fragments. We generate model fragment predicates using the following coverage criteria to combine partitions (cartesian product of partitions):
  – **AllRanges Criteria:** AllRanges specifies that each range in the partition of each property must be covered by at least one test model.
  – **AllPartitions Criteria:** AllPartitions specifies that the whole partition of each property must be covered by at least one test model.

The notion of coverage criteria to generate model fragments was initially proposed in our paper [5]. The accompanying tool called Meta-model Coverage Checker (MMCC) [5] generates model fragments using different test criteria taking any metamodel as input. Then, the tool automatically computes the coverage of a set of test models according to the generated model fragments. If some fragments are not covered, then the set of test models should be improved in order to reach a better coverage.

In this paper, we use the model fragments generated by MMCC for the UMLCD Ecore model (Figure 2). We use the criteria AllRanges and AllPartitions. For example, in Table 2, *mfAllRanges1* and *mfAllRanges2* are model fragments generated by PRAMANA using MMCC [5] for the *name* property of a classifier object. The *mfAllRanges1* states that there must be at least one classifier object with an empty name while *mfAllRanges2* states that there must be at least one classifier object with a non-empty name. These values for name are the ranges for the property. The model fragments chosen using AllRanges *mfAllRanges1* and *mfAllRanges2* define two partitions *partition1* and *partition2*. The model fragment *mfAll-*

Partitions1 chosen using AllPartitions defines both *partition1* and *partition2*.

These model fragments are transformed to ALLOY predicates by PRAMANA. For instance, model fragment *mfAllRanges7* is transformed to the predicate in Listing 3.

```
pred mfAllRanges7
{
  some c : Class | #c.attrs=1
}
```

Listing 3 ALLOY Predicate for *mfAllRanges7*

As mentioned in our previous paper [5] if a test set contains models where all model fragments are contained in at least one model then we say that the input domain is completely covered. However, these model fragments are generated considering only the concepts and relationships in the Ecore model and they do not take into account the constraints on the Ecore model. Therefore, not all model fragments are consistent with the input metamodel because the generated models that contain these model fragments do not satisfy the constraints on the metamodel. PRAMANA invokes the ALLOY Analyzer [17] to automatically check if a model containing a model fragment and satisfying the input domain can be synthesized for a general scope of number of objects. This allows us to *detect inconsistent model fragments*. For example, the following predicate, *mfAllRanges7a*, is the ALLOY representation of a model fragment specifying that some Class object does not have any Property object. PRAMANA calls the ALLOY API to execute the run statement for the predicate *mfAllRanges7a* along with the base ALLOY model to create a model that contains up to 30 objects per class/concept/signature (see Listing 4).

Table 2 Consistent Model Fragments Generated using AllRanges and AllPartitions Strategies

Model-Fragment	Description
mfAllRanges1	A Classifier $c \mid c.name =$""
mfAllRanges2	A Classifier $c \mid c.name! =$""
mfAllRanges3	A Class $c \mid c.is\_persistent = True$
mfAllRanges4	A Class $c \mid c.is\_persistent = False$
mfAllRanges5	A Class $c \mid \#c.parent = 0$
mfAllRanges6	A Class $c \mid \#c.parent = 1$
mfAllRanges7	A Class $c \mid \#c.attrs = 1$
mfAllRanges8	A Class $c \mid \#c.attrs > 1$
mfAllRanges9	An Attribute $a \mid a.is\_primary = True$
mfAllRanges10	An Attribute $a \mid a.name =$""
mfAllRanges11	An Attribute $a \mid a.name! =$""
mfAllRanges12	An Attribute $a \mid \#a.type = 1$
mfAllRanges13	An Association $as \mid as.name =$""
mfAllRanges14	An Association $as \mid \#as.src = 1$
mfAllRanges15	An Association $as \mid \#as.dest = 1$
mfAllPartitions1	Classifiers $c1, c2 \mid c1.name =$"" and $c2.name! =$""
mfAllPartitions2	Classes $c1, c2 \mid c1.is\_persistent = True$ and $c2.is\_persistent = False$
mfAllPartitions3	Classes $c1, c2 \mid \#c1.parent = 0$ and $\#c2.parent = 1$
mfAllPartitions4	Propertys $a1, a2 \mid a1.is\_primary = True$ and $a2.is\_primary = False$
mfAllPartitions5	Associations $as1, as2 \mid as1.name =$"" and $as2.name! =$""

```
pred mfAllRanges7a
{
  some c:Class | #c.attrs = 0
}

run mfAllRanges7a for 30
```

Listing 4 ALLOY Predicate and Run Command

The ALLOY analyzer yields a *no solution* to the run statement indicating that the model fragment is not consistent with the input domain specification. This is because no model can be created with this model fragment that also satisfies an input domain constraint that states that every Class must have at least one Property object as shown in Listing 5.

```
sig Class extends Classifier
{ ...
  attrs : some Attribute
  ...
}
```

Listing 5 Example ALLOY Signature

In Listing 5, *some* indicates 1..*. However, if a model solution can be found using ALLOY we call it a *consistent model fragment*. MMCC generates a total of 15 consistent model fragments using AllRanges and 5 model fragments using the AllPartitions strategy, as shown in Table 2.

3.4 Qualifying Models: Mutation Analysis for Model Transformation Testing

We generate sets of test models using different strategies and qualify these sets via mutation analysis [7]. Mutation analysis involves creating a set of faulty versions or *mutants* of a program. A test set must distinguish the program output from all the output of its mutants. In practice, faults are modelled as a set of mutation operators where each operator represents a class of faults. A mutation operator is applied to the program under test to create each mutant. A mutant is killed when at least one test model detects the pre-injected fault. It is detected when program output and mutant output are different. A test set is relatively adequate if it kills all mutants of the original program. A mutation score is associated to the test set to measure its effectiveness in terms of percentage of the killed/revealed mutants.

We use the mutation analysis operators for model transformations presented in our previous work [8]. These mutation operators are based on three abstract operations linked to the basic treatments in a model transformation: the navigation of the models through the relations between the classes, the filtering of collections of objects, the creation and the modification of the elements of the output model. Using this basis we define several mutation operators that inject faults in model transformations:

Relation to the same class change (RSCC): The navigation of one association toward a class is replaced with the navigation of another association to the same class.

Relation to another class change (ROCC): The navigation of an association toward a class is replaced with the navigation of another association to another class.

Relation sequence modification with deletion (RSMD): This operator removes the last step off from a navigation which successively navigates several relations.

Relation sequence modification with addition (RSMA): This operator does the opposite of RSMD, adding the navigation of a relation to an existing navigation.

Collection filtering change with perturbation (CFCP): The filtering criterion, which could be on a property or the type of the classes filtered, is disturbed.

Collection filtering change with deletion (CFCD): This operator deletes a filter on a collection; the mutant operation returns the collection it was supposed to filter.

Collection filtering change with addition (CFCA): This operator does the opposite of CFCD. It uses a collection and processes an additional filtering on it.

Class compatible creation replacement (CCCR): The creation of an object is replaced by the creation of an instance of another class of the same inheritance tree.

Classes association creation deletion (CACD): This operator deletes the creation of an association between two instances.

Classes association creation addition (CACA): This operator adds a useless creation of a relation between two instances.

Using these operators, we produced two hundred mutants from the class2rdbms model transformation with the repartition indicated in Table 3.

In general, not all mutants injected become faults as some of them are equivalent and can never be detected. The controlled experiments presented in this paper uses mutants presented in our previous work [8]. We have clearly identified faults and equivalent mutants to study the effect of our generated test models.

4 Methodology

We outline the methodology for test generation using PRAMANA and empirical evaluation of the generated test models via mutation analysis in Figure 3. The methodology uses the foundational ideas we present in Section 3 into a workflow.

Concisely, the test model generation workflow follows the steps:

Step 1: PRAMANA transforms input metamodel MM_{in}, its invariants I_{in}, the transformation pre-condition $pre(MT)$ of transformation MT and test strategy T to an ALLOY model (details in Sections 3.2, 3.3). PRAMANA solves the ALLOY model to generate instances that are in the input domain of MT.

Step 2: We try to execute all test models generated by PRAMANA as input to MT.

Step 3: If all test models are not executable in Step 2, then we incrementally create new pre-conditions for MT called $pre(MT)'$. These pre-conditions are created from rejected test models. We describe incremental pre-condition improvement in Section 4.1. We go to Step 1. Step 1 is executed using the a new source of constraints coming from $pre(MT)'$.

Step 4: If all test models are executable in Step 2, we perform mutation analysis using the generated test models with respect to the model transformation MT. Mutation analysis is described in Section 3.4.

4.1 Incremental Pre-condition Improvement

The execution of a transformation helps us discover new pre-condition constraints $pre(MT)'$ for the transformation MT. In this sub-section we present the approach to systematically create new pre-conditions.

Step 1: We execute the model transformation MT with a test model t. **Step 2:** The test model t is rejected by the model transformation and the transformation terminates. The rejection of a test model is often indicated by the raising of an exception by the transformation language. Some of the key sources for such rejections are:

1. **Insufficient memory** while creating modelling elements.
2. **Infinite Loop** while navigating a test model. For instance, this situation occurs when input models are navigated through a series of associations that can create loop structure in the transformation class2rdbms. These loops structures can navigation through diverse concepts such as inheritance trees, associations, and type of attributes. The Kermeta interpreter throws an StackOverflowError exception when it detects such a problem.
3. **Transformation of an input model to an output model not in output domain**. For instance, output models that do not satisfy the output meta-model specification and the post-condition $post(MT)$. In our case study, the transformation class2rdbms can produce ill-formed RDBMS models. A typical example is when a table contains several columns with same name. We detect these inconsistencies by checking if output models conform to the output meta-model (Ecore model of the meta-model with invariants) and satisfy post-conditions of the model transformation. The Figure 4 illustrates this detection. It represents an excerpt (bottom part) of an output model produced by the original transformation of a generated (excerpt on the top part).

Step 3: We isolate inconsistent output models and corresponding test models. We then use a traceability mechanism and tool such as in [18] to restrain the analysis of these models on excerpts such as the one illustrated in Figure 4. Class named *A* is transformed into one table because it is persistent. It redefined an association of the Class *B*. Two associations with the same name *asso1* point to classes with the same attribute/property *att1*. Respecting the specification, the original transformations produces a table with two columns named *asso1_att1*. This does not conform to the RDBMS metamodel and it is detected by our tool. Construction of such models can be prevented by generating objects with different names.

Table 3 Repartition of the class2rdbms mutants depending on the mutation operator applied

Mutation Operator	CFCA	CFCD	CFCP	CACD	CACA	RSMA	RSMD	ROCC	RSCC	Total
Number of Mutants	19	18	38	11	9	72	12	12	9	200

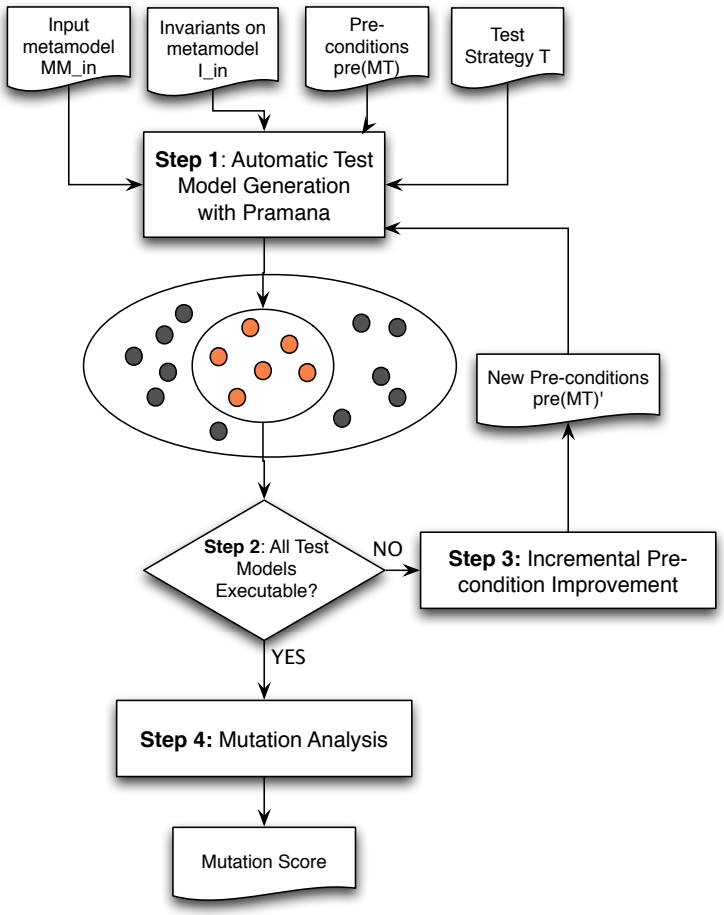

Fig. 3 Methodology for Automatic Test Generation and Mutation Analysis

Step 4: We solve this inconsistency by creating a new pre-condition constraint that protects the transformation from executing such models. We also regenerate new models that satisfy the new pre-condition constraints. For instance, the faulty model excerpt in Figure 4 can help us produce a new pre-condition that states:

In the classes of an inheritance tree, two associations with the same name can't point to classes that have (or their parent) attributes with same names.

Several new pre-conditions were discovered for the class2rdbms case study. We enlist nine newly discovered ALLOY facts in Appendix C apart from the initial set of pre-condition constraints as shown in Appendix B. These ALLOY facts can be easily expressed in OCL to improve the pre-condition specification of class2rdbms. The conditions may even be applicable to commercial implementations of class2rdbms.

5 Experiments

5.1 Experimental Setup and Execution

We use the methodology in Section 4 to compare coverage based test generation with unguided/random test model generation.

Coverage based test strategies as previously introduced in Section 3.3 consist of two test criteria AllRanges and AllPartitions. These test criteria generate model fragments from an effective input meta-model. A test set satisfying AllRanges must contain test models that contain all consistent model fragments from the AllRanges criteria. Similarly, a test set satisfying AllPartitions must contain all consistent model fragments generated from the AllPartitions criteria.

We generate sets of test models based on factorial experimental design [19]. We consider the *exact number of objects for each class* in the effective input meta-model as factors for

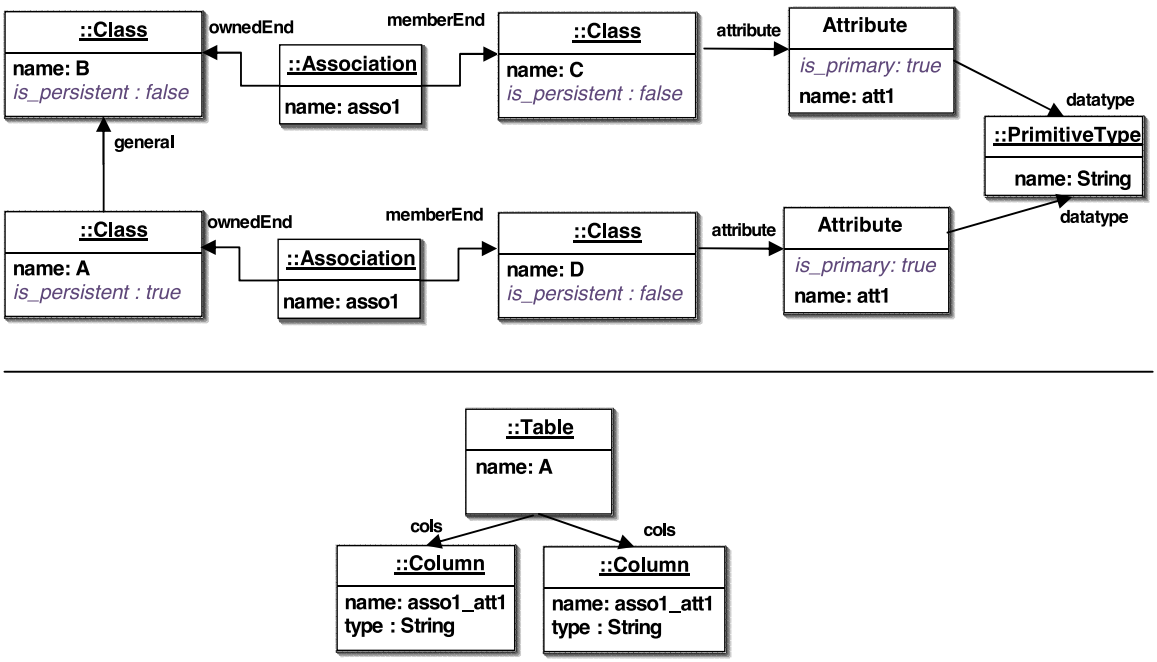

Fig. 4 Model Excerpt for Pre-condition Improvement

experimental design. A factor level is the exact number of objects of a given class in a test model. These factors help study the effect of number of different types of objects on the mutation score. For instance, we can ask questions such as whether a large number of Association objects have a correlation with the mutation score? The large of number Association objects also indicates a highly connected UML class diagram test model. We decide these factor levels by simple experimentation such as verifying if models can be generated in reasonable amount of time given that we need to generate thousands of test models in a few hours. We also want to cover a combination of a large number of varying factor levels. We have 8 different factor levels for the different classes in the UML class diagram effective input meta-model as shown in Table 4. Other factors that may affect but are not considered for test model generation are the use different SAT solvers such as SAT4J, MiniSAT, or ZChaff, maximum time to solve, t-wise interaction between model fragments.

The AllRanges criteria on the UMLCD meta-model gives 15 consistent model fragments (see Table 2). We have 150 models in a set, where 10 non-isomorphic models satisfies each different model fragment. We generate 10 non-isomorphic models to verify that mutation scores do not drastically change within each solution. We synthesize 8 sets of 150 models using different levels for factors as shown in Table 4 (see rows 1,2,3,4,5,6). The total number of models in these 8 sets is 1200.

The AllPartitions criteria gives 5 consistent model fragments. We have 50 test models in a set, where 10 non-isomorphic test models satisfies a different model fragment. We synthesize 8 sets of 50 models using factor levels shown in Table 4. The levels for factors for AllRanges and AllPartitions are the same. Total number of models in the 8 sets is 400. The selection of these factors at the moment is not based on a problem-independent strategy.

Table 4 Factors and their Levels for Test Sets

Factors	S1	S2	S3	S4	S5	S6	S7	S8
#ClassModel	1	1	1	1	1	1	1	1
#Class	5	5	15	15	5	15	5	15
#Association	5	15	5	15	5	5	15	15
#Attribute	25	25	25	25	30	30	30	30
#PrimitiveDataType	4	4	4	4	4	4	4	4
Bit-width Integer	5	5	5	5	5	5	5	5
#Models/Set AllRanges	15	15	15	15	15	15	15	15
#Models/Set Unguided	15	15	15	15	15	15	15	15
#Models/Set AllPartitions	5	5	5	5	5	5	5	
#Models/Set Unguided	5	5	5	5	5	5	5	

We compare test sets generated using AllRanges and AllPartitions with unguided test sets. For each test set of coverage based strategies we generate an equal number of random/unguided models as a reference to qualify the efficiency of different strategies. Precisely, we have 8 sets of 150 unguided test models to compare with AllRanges and 8 sets of 50 unguided test models to compare with AllPartitions. We use the factor levels in Table 4.

To summarize, we generate a total of 3200 models using an Intel(R) Core^{TM} 2 Duo processor with 4GB of RAM. We perform mutation analysis of these sets to obtain mutation scores on a grid of 10 Intel Celeron 440 high-end computers. The computation time for generating 3200 models was about

Table 5 Mutation Scores in Percentage for All Test Model Sets

Set	1	2	3	4	5	6	7	8
Unguided **150 models/set in 8 sets**	68.56	69.9	68.04	70.1	70.1	68.55	69	70.1
AllRanges **150 models/set in 8 sets**	88.14	92.26	81.44	85	91.23	80.4	91.23	88.14
Unguided **50 models/set in 8 sets**	70.1	62.17	68.04	70.1	65.46	68.04	69.94	70.1
AllPartitions **50 models/set in 8 sets**	90.72	93.3	84.53	87.62	87.62	82.98	92.78	88.66

3 hours and mutation analysis took about 1 week. We discuss the results of mutation analysis in the following section.

5.2 Results and Discussion

Mutation scores for AllRanges test sets are shown in Table 5 (row 2). Mutation scores for test sets obtained using AllPartitions are shown in Table 5 (row 4). We discuss the effects of the influencing factors on the mutation score:

- The number of Class objects and Association objects has a strong correlation with the mutation score. There is an increase in mutation score with the level of these factors. This is true for sets from unguided and model fragments based strategies. For instance, the lowest mutation score using AllRanges is 80.41 %. This corresponds to set 1 where the factor levels are 1,5,5,25,4,5 (see Column for set 1 in Table 4) and highest mutation scores are 91,24 and 92,27% where the factor levels are 1,15,5,25,4,5 and 1,5,15,25,4,5 respectively (see Columns for set 3 and set 7 in Table 4).
- We observe that AllPartitions test sets containing only 50 models/set gives a score of maximum 93.3%. The AllPartitions strategy demonstrates that knowledge from two different partitions satisfied by one test model greatly improves bug detecting efficiency. This also opens a new research direction to explore: Finding strategies to combine model fragments to guide generation of smaller sets of complex test models with better bug detecting effectiveness.

We compare unguided test sets with model fragment guided sets in the *box-whisker* diagram shown in Figure 5. The box whisker diagram is useful to visualize groups of numerical data such as mutation scores for test sets. Each box in the diagram is divided into lower quartile (25%), median, upper quartile (75% and above), and largest observation and contains statistically significant values. A box may also indicate which observations, if any, might be considered outliers or whiskers. In the box whisker diagram of Figure 5 we shown 4 boxes with whiskers for unguided sets and sets for AllRanges and AllPartitions. The X-axis of this plot represents the strategy used to select sets of test models and the Y-axis represents the mutation score for the sets.

We make the following observations from the box-whisker diagram:

- Both the boxes of AllRanges and AllPartitions represent mutation scores higher than corresponding unguided sets.
- The high median mutation scores for strategies AllRanges 88.14% and AllPartitions 88.14% indicate that both these strategies return consistently good test sets.
- The small size of the box for AllPartitions compared to the AllRanges box indicates its relative convergence to good sets of test models.
- The small set of 50 models using AllPartitions gives mutations scores equal or greater than 150 models/set using AllRanges. This implies that it is a more efficient strategy for test model selection. The main consequence is a reduced effort to write corresponding *test oracles* [20] with 50 models compared to 150 models.
- Despite the generation of multiple solutions (10 solutions for each model fragment or an empty fragment for unguided generation) for each strategy we see a consistent behaviour in the mutation scores. There is no large difference in the mutation scores especially for unguided generation. The median is 69% and the mutation scores range between 68% and 70%. The AllRanges and AllPartitions vary a little more in their mutation scores due to a larger coverage of the effective input meta-model.

The freely and automatically obtained knowledge from the input meta-model using the MMCC algorithm shows that AllRanges and AllPartitions are successful strategies to guide test generation. They have higher mutation scores with the same sources of knowledge used to generate unguided test sets. A manual analysis of the test models reveals that injection of inheritance via the parent relation in model fragments results in higher mutation scores. Most unguided models do not contain inheritance relationships as it is not imposed by the meta-model.

What about the 7% of the mutants that remain alive given that the highest mutation score is 93.3%? We note by an analysis of the live mutants that they are the same for both AllRanges and AllPartitions. There remain 19 live mutants in a total of 200 injected mutants (with 6 equivalent mutants). In the median case both AllRanges and AllPartitions strategy give a mutation score of 88.14%. The live mutants in the median case are mutants not killed due to fewer objects in models.

To consistently achieve a higher mutation score we need more CPU speed, memory and parallelization to efficiently generate larger test models and perform mutation analysis on them. This extension of our work has not be been explored in the paper. It is important for us to remark that some live mutants can only be killed with more information about the model transformation such as those derived from its requirements specification. For instance, one of the remaining live mutant requires a test model with a class containing several

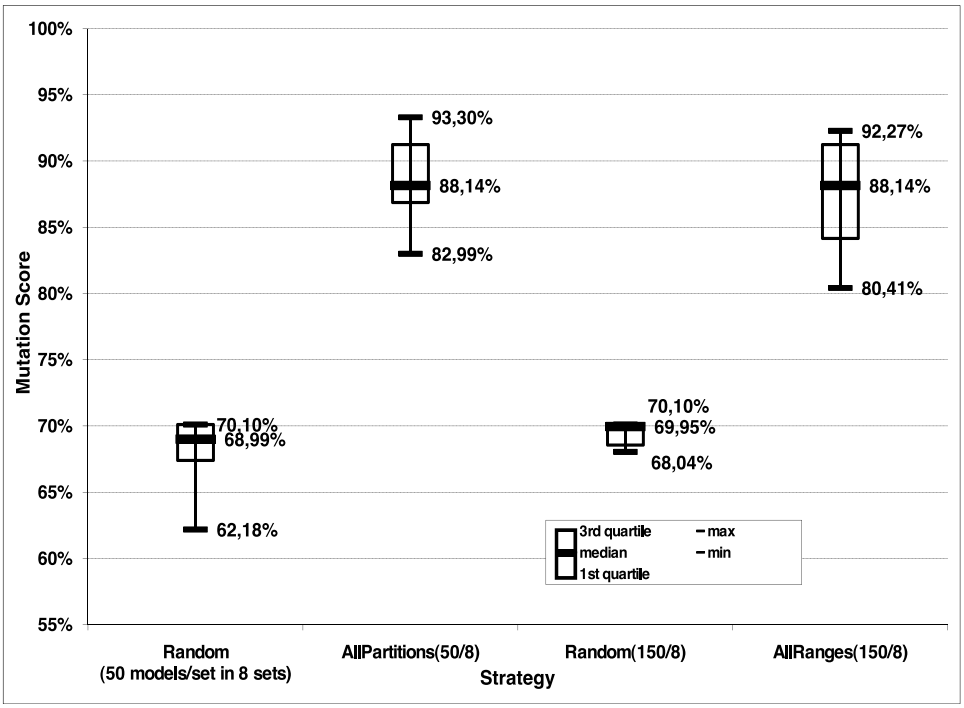

Fig. 5 Box-whisker Diagram to Compare Automatic Model Generation Strategies

primitive type attributes such that at least one is a primary attribute. A test model that satisfies such a requirement requires the combination of model fragments imposing the need for several attributes in a class A, attributes of class A must have primitive types, at least one primary attribute in the class A, and at least one non-primary attribute in the class A. This requirement can either be specified by manually creating a combination of fragments or by developing a better general test strategy to combine multiple model fragments. In another situation, we observe that not all model fragments are consistent with the input domain and hence they do not really cover the entire meta-model. Therefore, we miss killing some mutants. This information could help improve partitioning and combination strategies to generate better test sets.

6 Related Work

We explore three main areas of related work : test criteria, automatic test generation, and qualification of strategies.

The first area we explore is work on test criteria in the context of model transformations in MDE. Random generation and input domain partitioning based test criteria are two widely studied and compared strategies in software engineering (non MDE) [21] [22] [23]. To extend such test criteria to MDE we have presented in [5] input domain partitioning of input meta-models in the form of model fragments. However, there exists no experimental or theoretical study to qualify the approach proposed in [5].

Experimental qualification of the test strategies require techniques for automatic model generation. Model generation is more general and complex than generating integers, floats, strings, lists, or other standard data structures such as dealt with in the Korat tool of Chandra et al. [24]. Korat is faster than ALLOY in generating data structures such as binary trees, lists, and heap arrays from the Java Collections Framework but it does not consider the general case of models which are arbitrarily constrained graphs of objects. The constraints on models makes model generation a different problem than generating test suites for context-free grammar-based software [25] which do not contain domain-specific constraints.

Test models are complex graphs that must conform to an input meta-model specification, a transformation pre-condition and additional knowledge such as model fragments to help detect bugs. In [26] the authors present an automated generation technique for models that conform only to the class diagram of a meta-model specification. A similar methodology using graph transformation rules is presented in [27]. Generated models in both these approaches do not satisfy the constraints on the meta-model. In [28] we present a method to generate models given partial models by transforming the meta-model and partial model to a Constraint Logic Programming (CLP). We solve the resulting CLP to give model(s) that conform to the input domain. However, the approach does not add new objects to the model. We assume that the number and types of models in the partial model is sufficient for obtaining complete models. The constraints in this system are limited to first-order horn clause logic. In [6] we have introduce a tool CARTIER based on the constraint solving system ALLOY to resolve the issue of generating models such that constraints over both objects and properties are satisfied simultaneously. In this paper we use CARTIER to systemati-

cally generate several hundred models driven by knowledge/-constraints of model fragments [5]. Statistically relevant test model sets are generated from a factorial experimental design [19] [29].

The qualification of a set of test models can be based on several criteria such as code and rule coverage for white box testing, satisfaction of post-condition or mutation analysis for black/grey box testing. In this paper we are interested in obtaining the relative adequacy of a test set using mutation analysis [7]. In previous work [8] we extend mutation analysis to MDE by developing mutation operators for model transformation languages. We qualify our approach using a representative transformation UMLCD models to RDBMS models called class2rdbms implemented in the transformation language Kermeta [3]. This transformation [11] was proposed in the MTIP Workshop in MoDeLs 2005 as a comprehensive and representative case study to evaluate model transformation languages.

7 Conclusion

Testing model transformations presents the challenging problem of developing approaches to automatically generate effective test models. In this paper we present PRAMANA, a tool to generate models conforming in the input domain of a transformation and guided by different coverage criteria. First, PRAMANA helps us precisely specify the input domain of a model transformation incremental pre-condition improvement. Second, we use PRAMANA to generate sets of test models that compare coverage and unguided strategies for model generation. All test sets using these strategies detect faults given by their mutation scores. The comparison of coverage strategies with unguided generation taught us that both strategies AllPartitions and AllRanges look very promising. Coverage strategies give a maximum mutation score of 93% compared to a maximum mutation score of 70% in the case of unguided test sets. We observe that mutation scores do not vary drastically despite the generation of multiple solutions for the same test strategy. We conclude from our experiments that the AllPartitions strategy is a promising strategy to consistently generate a small test of test models with a good mutation score. However, to improve effectiveness of test sets we might require effort from the test designer to obtain test model knowledge/test strategy that take the internal model transformation design requirements into account. The experiments in this paper were performed based on the mutation analysis of class2rdbms written in the language Kermeta. In future, we intend to develop mutation analysis tools for various mature model transformation languages. The automatic mutation analysis tool will help us perform experiments using a number of transformation case studies. Applying our approach to large input metamodels such as the UML is a challenge in scaling our approach. We intend to leverage our recently developed technique called *metamodel pruning* [30] to extract a small subset of large metamodel such as UML which is conducive to constraint solving in PRAMANA and consequently model generation for large input metamodels.

A ALLOY Model Synthesized by CARTIER

```
module tmp/UMLCD
open util/boolean as Bool

sig Model
{
  classifier : set Classifier,
  association:set Association
}


abstract sig Classifier
{
  name :  Int
}

sig PrimitiveDataType extends Classifier
{ }

sig Class extends Classifier
{
  is_persistent: one Bool,
  parent : lone Class,
  attrs : some Attribute

}

sig Association

{
  name: Int,
  dest: one Class,
  src: one Class

}

sig Attribute
{
  name: Int,
  is_primary : Bool,
  type: one Classifier
}

//Meta-model constraints

/* There must be No Cyclic Inheritance in an UMLCD*/

fact noCyclicInheritance
{
  no c: Class | c in c.^parent
}

/* All the attributes in a Class must have unique attribute names */

fact uniqueAttributeNames
{
  all c:Class | all a1:  c.attrs, a2: c.attrs | a1.name = a2.name implies a1=a2
}

/* An attribute object can be contained by only one class */

fact attributeContainment
{
  all c1:Class, c2:Class | all a1: c1.attrs , a2 : c2.attrs | a1 = a2 implies c1=
      c2
}

/* There is exactly one Model object */

fact oneModel
{
  #Model=1
}

/*All Classifier objects are contained in a Model*/

fact classifierContainment
{
  all c:Classifier | c in Model.classifier
}

/*All Association objects are contained in a Model */

fact associationContainment
{
all a:Association| a in Model.association
}

/*A Classifier must have a unique name in the Class Diagram*/

fact uniqueClassifierName
{
  all c1:Classifier, c2:Classifier |c1.name = c2.name implies c1=c2
}

/*An associations have the same name either they are the same association or they
      have different sources*/

fact uniqeNameAssocSrc
{
  all a1:Association, a2:Association |
  a1.name = a2.name implies (a1 = a2 or a1.src != a2.src)
}
```

Listing 6 ALLOY Model for UML Class Diagram

B Initial Set of Pre-conditions

```
/*Initial Model Transformation Pre-conditions*/


fact atleastOnePrimaryAttribute
{
  all c:Class | one a:c.attrs | a.is_primary =True
}

fact no4CyclicClassAttribute
{
  all a:Attribute | a.type in Class implies all a1:a.type.attrs | a1.type in
  Class implies all a2:a.type.attrs | a2.type in Class implies all a3:a.type.
       attrs|a3.type
   in Class implies all a4:a.type.attrs | a4.type in PrimitiveDataType
}

fact noAttributeAndAssociationHaveSameName
{
  all c:Class , assoc :Association |
  all a:c.attrs | (assoc.src = c) implies a.name != assoc.name
}

fact no1CycleNonPersistent
{
all a: Association | (a.dest = a.src) implies a.src.is_persistent = True
}

fact no2CycleNonPersistent
{
  all a1: Association, a2:Association |
  (a1.dest = a2.src and a2.dest = a1.src) implies
  a1.src.is_persistent = True or a2.src.is_persistent=True
}
```

Listing 7 Initial pre-conditions as ALLOY facts

C Discovered Set of Pre-conditions

```
//Discovered Model Transformation pre-condition constraints

/* 1. At a depth of 4 the type of an attribute has to be primitive  and cannot be
       a class type*/

fact no4CyclicClassAttribute{
  all a:Attribute | a.type in Class => all a1:a.type.attrs|a1.type in
  Class => all a2:a.type.attrs|a2.type in Class => all
  a3:a.type.attrs|a3.type in Class => all a4:a.type.attrs|a4.type
  in PrimitiveDataType }

/* 2.  A Class cannot have an association and an attribute of the same name */

fact noAttribAndAssocSameName{
  all c:Class, assoc:Association | all a : c.attrs | (assoc.src == c) => a.name
       != assoc.name
}


/* 3. No cycles between non-persistent classes */

fact no1CycleNonPersistent
{
      all a: Association | (a.dest == a.src) => a.dest.is_persistent= True
}



fact no2CycleNonPersistent
{
      all a1: Association, a2:Association | (a1.dest == a2.src and a2.dest==a1.
           src) => a1.src.is_persistent= True or a2.src.is_persistent=True
}



/* 4.  A persistent class can't have an association to one of its parent */

fact assocPersistentClass
{
  all a:Association | a.src.is_persistent=True implies a.dest not in a.src.^
       parent
}



/* 5. Unique association names in a class hierarchy */

fact uniqueAssocNamesInInHeritanceTree
{
     all c:Class |
    all a1:Association, a2:Association |
    (a1.src in c and a2.src in c.^parent and a1!=a2) implies (a1.name!=a2.name)


 }


/* 6. A class can't be the type of one of its attributes (amoung all its
attributes */
```

```
fact classCantTypeOfAllofItsAttribute
{
 all c:Class | all a: (c.attrs+c.^parent.attrs) | a.type !=c
}

/* 7. A Class A which inherits from a persistent class B can't have an outgoing
       association with the same name
that one association of that persistent class B */

fact classInheritsOutgoingNotSameNameAssoc
{
  all A:Class | all B:A.^parent | B.is_persistent == True implies (no a1:
       Association, a2:Association |
(a1.src = A and a2.src=B and a1.name=a2.name))
}

/* 8. A class A which inherits from a persistent class B can't have an attribute
       with the same name
that one attribute of that persistent class B */


fact classInheritsOutgoingNotSameNameAttrib
{
  all A:Class | all B:A.^parent | B.is_persistent == True implies (no a1: A.attrs
       , a2:B.attrs |
(a1.name=a2.name))
}

/* 9. No association between two classes of an inheritance tree */

fact noAssocBetweenClassInHierarchy
{
  all a : Association | all c: Class | (a.src =c implies a.dest not in c.^parent)
       and (a.dest =c implies a.src not in c.^parent)
}
```

Listing 8 Discovered pre-conditions as ALLOY facts

References

1. Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., Mottu, J.M.: Barriers to systematic model transformation testing. Communications of the ACM (2009)
2. Bardohl, R., Taentzer, G., M. Minas, A.S.: Handbook of Graph Grammars and Computing by Graph transformation, vII: Applications, Languages and Tools. World Scientific (1999)
3. Muller, P.A., Fleurey, F., Jezequel, J.M.: Weaving executability into object-oriented meta-languages. In: Inernational Conference on Model Driven Engineering Languages and Systems (MoDelS/UML), Montego Bay, Jamaica, Springer (2005) 264–278
4. Jouault, F., Kurtev, I.: On the Architectural Alignment of ATL and QVT. In: Proceedings of ACM Symposium on Applied Computing (SAC 06), Dijon, FRA (April 2006)
5. Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.L.: Towards dependable model transformations: Qualifying input test data. Software and Systems Modelling (Accepted) (2007)
6. Sen, S., Baudry, B., Mottu, J.M.: On combining multi-formalism knowledge to select test models for model transformation testing. In: IEEE International Conference on Software Testing, Lillehammer, Norway (April 2008)
7. DeMillo, R., R.L., Sayward, F.: Hints on test data selection : Help for the practicing programmer. IEEE Computer **11**(4) (1978) 34 – 41
8. Mottu, J.M., Baudry, B., Traon, Y.L.: Mutation analysis testing for model transformations. In: Proceedings of ECMDA'06, Bilbao, Spain (July 2006)
9. Budinsky, F.: Eclipse Modeling Framework. The Eclipse Series. Addison-Wesley (2004)
10. OMG: The Object Constraint Language Specification 2.0, OMG Document: ad/03-01-07 (2007)
11. Bezivin, J., Rumpe, B., Schurr, A., Tratt, L.: Model transformations in practice workshop, october 3rd 2005, part of models 2005. In: Proceedings of MoDELS. (2005)
12. OMG: Mof 2.0 core specification. Technical Report formal/06-01-01, OMG (April 2006) OMG Available Specification.

13. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: MODELS/UML. Volume 3713., Montego Bay, Jamaica, Springer (October 2005) 264–278
14. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: Uml2alloy: A challenging model transformation. In: MoDELS. (2007) 436–450
15. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from uml to alloy. In: Software and Systems Modeling. Volume 9. (December 2009) 69–86
16. Baar, T.: The definition of transitive closure with ocl-limitations and applications. In: Proceedings of Fifth Andrei Ershov International Conference, Perspectives, of System Informatics, Springer (2003) 358–365
17. Jackson, D.: http://alloy.mit.edu. (2008)
18. Glitia, F., Etien, A., Dumoulin., C.: Traceability for an mde approach of embedded system conception. in ,. In: Fourth ECMDA Tracibility Workshop, Berlin, Germany (June 2008)
19. Pfleeger, S.L.: Experimental design and analysis in software engineering. Annals of Software Engineering (2005) 219–253
20. Mottu, J.M., Baudry, B., Traon, Y.L.: Model transformation testing: Oracle issue. In: In Proc. of MoDeVVa workshop colocated with ICST 2008, Lillehammer, Norway (April 2008)
21. Vagoun, T.: Input domain partitioning in software testing. In: HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS) Volume 2: Decision Support and Knowledge-Based Systems, Washington, DC, USA (1996)
22. Weyuker, E.J., Weiss, S.N., Hamlet, D.: Comparison of program testing strategies. In: TAV4: Proceedings of the symposium on Testing, analysis, and verification, New York, NY, USA, ACM (1991) 1–10
23. Gutjahr, W.J.: Partition testing versus random testing: the influence of uncertainty. IEEE TSE **25** (1999) 661–674
24. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on java predicates. In: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis. (2002)
25. M, H., Power, J.: An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In: Proc. of the 20th IEEE/ACM ASE, NY, USA (2005)
26. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon., Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: Proceedings of ISSRE'06, Raleigh, NC, USA (2006)
27. Ehrig, K., Kster, J., Taentzer, G., Winkelmann, J.: Generating instance models from meta models. In: FMOODS'06 (Formal Methods for Open Object-Based Distributed Systems), Bologna, Italy (June 2006) 156 – 170.
28. Sen, S., Baudry, B., Precup, D.: Partial model completion in model driven engineering using constraint logic programming. In: International Conference on the Applications of Declarative Programming. (2007)
29. Federer, W.T.: Experimental Design: Theory and Applications. Macmillan (1955)
30. Sen, S., Moha, N., Baudry, B., Jezequel, J.M.: Meta-model pruning. In: Model Driven Engineering Languages and Systems, 12th International Conference (MODELS), Denver, CO, USA (October 4-9 2009)