Automatic Model Generation Strategies for Model Transformation Testing

Sagar Sen, Benoit Baudry, Jean-Marie Mottu

To cite this version:

Sagar Sen, Benoit Baudry, Jean-Marie Mottu. Automatic Model Generation Strategies for Model Transformation Testing. Second International Conference, ICMT 2009, Zurich, Switzerland, June 29-30, 2009., Jun 2009, Zurich, Switzerland. pp.148-164. inria-00468256

HAL Id: inria-00468256

https://hal.inria.fr/inria-00468256

Submitted on 30 Mar 2010

Automatic Model Generation Strategies for Model Transformation Testing

Sagar Sen[1], Benoit Baudry[1], and Jean-Marie Mottu[1]

INRIA, Centre Rennes - Bretagne Atlantique, Campus universitaire de beaulieu, Rennes Cedex 35000, France `ssen,bbaudry,jmottu@irisa.fr`

Abstract. Testing *model transformations* requires input models which are graphs of inter-connected objects that must conform to a meta-model and meta-constraints from heterogeneous sources such as well-formedness rules, transformation pre-conditions, and test strategies. Manually specifying such models is tedious since models must simultaneously conform to several meta-constraints. We propose automatic model generation via constraint satisfaction using our tool Cartier for model transformation testing. Due to the virtually infinite number of models in the input domain we compare strategies based on input domain partitioning to guide model generation. We qualify the effectiveness of these strategies by performing mutation analysis on the transformation using generated sets of models. The test sets obtained using partitioning strategies gives mutation scores of up to 87% vs. 72% in the case of unguided/random generation. These scores are based on analysis of 360 automatically generated test models for the representative transformation of UML class diagram models to RDBMS models.

1 Introduction

Model transformations are core MDE components that automate important steps in software development such as refinement of an input model, re-factoring to improve maintainability or readability of the input model, aspect weaving, exogenous and endogenous transformations of models, and generation of code from models. Although there is wide spread development of model transformations in academia and industry there is mild progress in the domain of validating transformations. In this paper, we address the challenges in validating model transformations via *black-box testing*. We think that black-box testing is an effective approach to validating transformations due to the diversity of transformation languages based on graph rewriting [1], imperative execution (Kermeta [2]), and rule-based transformation (ATL [3]) that render language specific formal methods and white-box testing impractical.

In black-box testing of model transformations we require *test models* that can detect bugs in the model transformation. These models are graphs of inter-connected objects that must conform to a meta-model and satisfy meta-constraints such as well-formedness rules and transformation pre-conditions.

Automatic model generation based on constraint satisfaction is one approach to ensure that meta-constraints and test requirements are simultaneously satisfied by models. In previous work [4], we introduce a tool Cartier that transforms the input meta-model (in Eclipse Model Framework (EMF) standard [5]) of a model transformation and pre-condition (in a textual language such as OCL [6]) to a common constraint language Alloy. Cartier invokes Alloy to generate a Boolean CNF formula and solve it using a

SAT solver [7] to obtain solutions at a low-level of abstraction. Cartier transforms these solutions back to instances of the high-level input meta-model (as XMI instances of the EMF meta-model). However, most models generated using Cartier are only trivial as they are not guided by a strategy. One of the goals of our work is to compare transformation independent strategies to guide automatic test model generation in order to detect bugs.

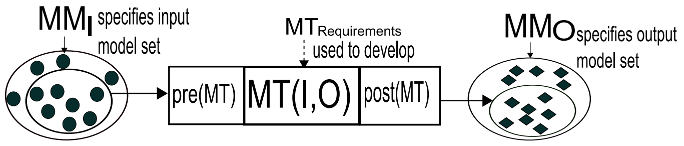
In this paper we use Cartier to systematically generate finite sets of models from the space of virtually infinite input models using different test strategies. First, we generate sets of random models (we call this strategy random for convenience although random or pseudorandom models is still not a well-defined concept and is not the focus of this paper). We also generate sets of models guided by model fragments obtained from meta-model partitioning strategies presented in our previous work [8]. We compare strategies using mutation analysis of model transformations [9] [10]. Mutation analysis serves as a *test oracle* to determine the relatively adequacy of generated test sets. We do not use domain-specific post-conditions as oracles to determine the correctness of the output models. We use the representative model transformation of Unified Modelling Language Class Diagram (UMLCD) to Relational Database Management Systems (RDBMS) models called class2rdbms to illustrate our model generation approach and effectiveness of test strategies. The mutation scores show that input domain partitioning strategies guide model generation with considerably higher bug detection abilities (87%) compared to unguided generation (72%). Our results are based on 360 generated test models and about 50 hours of computation on a high-end server. We summarize our contributions as follows:

– **Contribution 1:** We use Cartier [4] to generate hundreds of valid models using different search strategies.
– **Contribution 2:** We use mutation analysis [10] to compare sets of generated models for their bug detecting effectiveness. We show that model sets generated using partitioning strategies, previously presented in our work [8], help detect more bugs than unguided generation.

The paper is organized as follows. In Section 2 we present the transformation testing problem and the case study. In Section 3 we describe our tool Cartier for automatic model generation, strategies for guiding model generation, and mutation analysis for model transformation testing. In Section 5 we present the experimental methodology, setup and results to compare model generation strategies. In Section 6 we present related work. We conclude in Section 7.

2 Problem Description

We present the problem of black-box 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 meta-model MM_I (UMLCD in Figure 2). The set of all output models is specified by meta-model MM_O. The pre-condition of the model transformation $pre(MT)$ further constrains the input domain. A post-condition $post(MT)$ limits the model transformation to producing a subset of all possible output models. The model transformation is developed based on a set of requirements $MT_{Requirements}$.



MMI specifies input model set
MTRequirements used to develop
MMO specifies output model set
pre(MT)
MT(I,O)
post(MT)

Fig. 1. A Model Transformation

Model generation for black-box 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 meta-model and its invariants MM_I or the transformation pre-condition $pre(MT)$. However, in this paper we only focus on detecting bugs in a transformation.

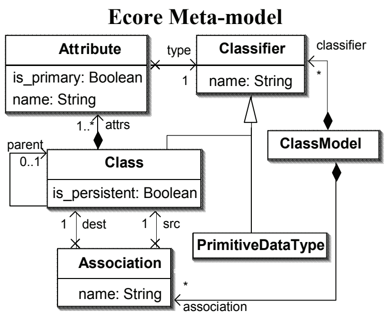
2.1 Transformation Case Study

Our case study is the transformation from 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 black-box testing we need input models that conform to the input meta-model 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 input meta-model for class2rdbms. The concepts and relationships in the input meta-model are stored as an Ecore model [5] (Figure 2 (a)). The invariants on the UMLCD Ecore model, expressed in Object Constraint Language (OCL) [6], are shown in Figure 2 (b). The Ecore model and the invariants together represent the input meta-model for class2rdbms. The OCL and Ecore are industry standards used to develop meta-models 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. In [11] the authors present some limitations of OCL.

The input meta-model 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 Attribute object (b) The type of an Attribute object can be a Class C, but finally the transitive closure of the type of Attribute objects of Class C must end with type PrimitiveDataType. In our case we approximate this recursive closure constraint by stating that Attribute 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 Attribute 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 [12] to experiment and validate model transformation language features. The input domain meta-model of UML class diagram model covers all major meta-modelling concepts



(a)

OCL Invariants

context Class

inv noCyclicInheritance:not self.allParents()->includes(self)

inv uniqueAttributesName:
 self.attrs->forAll(att1, att2 |att1.name=att2.name implies att1=att2)

context ClassModel

inv uniqueClassifierNames:
 self.classifier->forAll(c1, c2 |c1.name=c2.name implies c1=c2)

inv uniqueClassAssociationSourceName :
self.association->forAll(ass1, ass2 |
 ass1.name=ass2.name implies (ass1=ass2 or ass1.src != ass2.src))

(b)

Fig. 2. (a) Simple UML Class Diagram Ecore Model (b) OCL constraints on the Ecore model

such as inheritance, composition, finite and infinite multiplicities. The constraints on the UML meta-model 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 [10]) enabling us to test essential model transformation features. Among the limitations the simple version of the UMLCD meta-model does not contain Integer and Float attributes. The number of classes in the simplified UMLCD meta-model is not very high when compared to the standard UML 2.0 specification. There are also no inter meta-model references and arbitrary containments in the simple meta-model.

Model generation is relatively fast but performing mutation analysis is extremly time consuming. Therefore, we perform mutation analysis on class2rdbms to qualify transformation and meta-model 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 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 10% of the time while performing mutation analysis takes most of the time.

3 Automatic Model Generation

We use the tool Cartier previously introduced in our paper [4] to automatically generate models. We invoke Cartier to transform the input domain specification of a model transformation to a common constraint language Alloy. Then Cartier invokes the Alloy API to obtain Boolean CNF formulae [13], launch a SAT solver such as ZChaff [7] to generate models that conform to the input domain of a model transformation.

Cartier transforms a model transformation's input meta-model expressed in the Eclipse Modelling Framework [5] format called Ecore using the transformation rules presented in [4]. OCL constraints and natural language constraints on the input ecore meta-model are manually transformed to Alloy facts. These OCL constraints are used to express meta-model invariants and model transformation pre-conditions. We do not automate OCL to Alloy as there are several challenges posed by this transformation as 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. 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 meta-model to Alloy from their original OCL specifications. Identifying a subset of OCL that can be automatically transformed to Alloy is an *open challenge*. As an example transformation consider the invariant for no cyclic inheritance in Figure 2(b). The constraint is specified as the following fact:

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

The generated Alloy model for the the UMLCD meta-model is given in Appendix A. This Alloy model only describes the input domain of the transformation. Solving the facts and signatures in the model (see Section 3.2) results in *unguided and trivial solutions*. Are these trivial solution capable of detecting bugs? This is the question that is answered in Section 5. Are there better heuristics to generate test models? In the following sub-section we illustrate how one can guide model generation using strategies based on input domain partitioning.

3.1 Strategies to Guide Model Generation

Good strategies to guide automatic model generation are required to obtain 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 Cartier as described in Section 3. This combined Alloy model is solved and the solutions are transformed to model instances of the input meta-model that satisfy the predicate. We present the following strategies to guide model generation:

– **Random/Unguided Strategy:** The basic form of model generation is unguided where only the Alloy model obtained from the meta-model 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 $pred\, random\{\}$.
– **Input-domain Partition based Strategies:** We guide generation of models using test criteria to combine *partitions* on domains of all properties of a meta-model (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 meta-model we define two test criteria that are based on different strategies for combining partitions of properties. Each criterion defines a set of *model fragments* for an input meta-model. These fragments are transformed to predicates on meta-model properties by Cartier. 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 test 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 test criteria to generate model fragments was initially proposed in our paper [8]. The accompanying tool called Meta-model Coverage Checker (MMCC) [8] generates model fragments using different test criteria taking any meta-model 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 1, *mfAllRanges1* and *mfAllRanges2* are model fragments generated by Cartier using MMCC [8] 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 *mfAllPartitions1* chosen using AllPartitions defines both *partition1* and *partition2*.

Table 1. 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 $as \mid \#as.dest = 0$
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	Attributes $a1, a2 \mid a1.is\_primary = True$ and $a2.is\_primary = False$
mfAllPartitions5	Associations $as1, as2 \mid as1.name =$“” and $as2.name! =$“”

These model fragments are transformed to Alloy predicates by Cartier. For instance, model fragment mfAllRanges7 is transformed to the predicate :

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

As mentioned in our previous paper [8] 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 meta-model because the generated models that contain these model fragments do not satisfy the constraints on the meta-model. Cartier invokes the Alloy Analyzer [15] 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 Attribute object. Cartier 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:

```
pred mfAllRange7a(){some c:Class|#c.attrs=0}
run mfAllRanges7 for 30
```

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 Attribute object:

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

, where *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 1.

3.2 Model Generation by Solving Alloy Model

Given the base Alloy model with signatures, facts and predicates from model fragments (see A) Cartier synthesizes Alloy run commands. Cartier synthesizes a run command for a given scope or based on exact number of objects per class/signature. A scope is the maximum number of objects/atoms per signature. Scope can be specified for individual signatures or the same scope can apply to all signatures. Executing the following run command in Alloy attempts to generates a model that conforms to the input domain and satisfies the model fragment called mfAllRanges1.

```
run mfAllRanges1 for 20
```

Cartier invokes a SAT solver using the Alloy API to incrementally increase the scope unto 20 and see if one or more solutions can be found. If solutions can be found we transform the low-level Alloy XML output to XMI that can be read by Ecore based model transformations or editors. On the other hand, we can also specify the correct number of atoms/objects per signature as shown below.

```
run mfAllRanges1 for for 1 ClassModel,5 int, exactly 5 Class,
exactly  25 Attribute, exactly 4 PrimitiveDataType,exactly 5 Association
```

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 [9]. 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 [10]. 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.

Table 2. Repartition of the UMLCD2RDBMS 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

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 2.

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 [10]. We have clearly identified faults and equivalent mutants to study the effect of our generated test models.

5 Empirical Comparison of Generation Strategies

5.1 Experimental Methodology

We illustrate the methodology to qualify test generation strategies in Figure 3. The methodology flows is: (1) The inputs to Cartier are an Ecore meta-model, Alloy facts on the Ecore model, Alloy predicates for transformation pre-condition and experimental design parameters (such as factor levels, discussed shortly) (2) Cartier generates an Alloy model from the Ecore using rules in [4]. The input facts and predicates are inserted into the Alloy model. MMCC uses the Ecore to generate model fragments which Cartier transforms to Alloy predicates which are inserted into to the Alloy model. Cartier uses experiment design parameters to generate run commands and inserts them into the Alloy model. These aspects are inserted in the sequence: signatures, facts, predicate, and run commands (3) Cartier invokes the Alloy API to execute each run command (4) Cartier invokes run commands that uses the KodKod engine [13] in Alloy to transform the Alloy

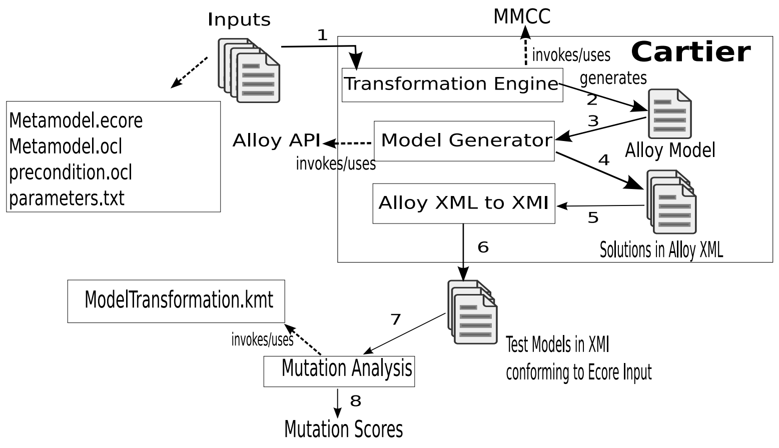


Fig. 3. Experimental Methodology to Qualify Automatic Model Generation Strategies

Table 3. Factors and their Levels for AllRanges and AllPartitions Test Sets

Factors	**Sets:** 1	2	3	4	5	6	7	8
#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 AllPartitions	5	5	5	5	5	5	5	

Table 4. Factor Levels for Random Pool of 200 Test Models

Factors	**Levels**
#ClassModel	1
#Class	5,10,15,20,25
#Association	5,10,15,20,25
#Attribute	25,30,35,40
#Primitive DataType	3,4
Bit-width Integer	5

model to Boolean CNF, followed by invocation of a SAT solver such as ZChaff [7] to solve the CNF and generate solutions in Alloy XML (5,6) Cartier transforms Alloy XML instances to XMI using the input Ecore meta-model (7) We obtain XMI models in different sets for different strategies. Mutation analysis is performed on each of these sets with respect to a model transformation to give a mutation score for each set (8) We represent the mutation scores in a box-whisker diagram to compare and qualify strategies.

5.2 Experimental Setup and Execution

We use the methodology in Section 5.1 to compare model fragment driven test generation with unguided/random test model generation. We consider two test criteria for generating model fragments from the input meta-model: AllRanges and AllPartitions. We compare test sets generated using AllRanges and AllPartitions with randomly generated test sets containing an equal number of models. We use experimental design [16] to consider the effect of different factors involved in model generation. We consider the exact number of objects for each class in the input meta-model as factors for experimental design. The AllRanges criteria on the UMLCD meta-model gives 15 consistent model fragments (see Table 1). We have 15 models in a set, where each model satisfies one different model fragment. We synthesize 8 sets of 15 models using different levels for factors as shown in Table 3 (see rows 1,2,3,4,5,6). The total number of models in these 8 sets is 120. The AllPartitions criteria gives 5 consistent model fragments. We have 5 test models in a set, where each model satisfies a different model fragment. We synthesize 8 sets of 5 models using factor levels shown in Table 3. The levels for factors for AllRanges and AllPartitions are the same. Total number of models in the 8 sets is 40. The selection of these factors at the moment is not based on a problem-independent strategy. They are chosen based on the capacity of the solver in obtaining a model with 100 to 200 objects for our case study in a reasonable amount of time.

We create random/unguided models as a reference to qualify the efficiency of different strategies. We generate a pool of 200 unguided/random test models. We select this pool of test models using all the unique combinations of factor levels shown in the Table 4. We then randomly select 15 models at a time from this pool to create 8 sets of random models. We use these sets to compare mutation scores of 8 sets we obtain for the AllRanges strategy. Similarly, we randomly select 5 models at a time from the pool

Table 5. Mutation Scores in Percentage for All Test Model Sets

Set	1	2	3	4	5	6	7	8
Random **15 models/set in 8 sets**	72.6	70.61	69	71.64	72.68	72.16	69	69
AllRanges **15 models/set in 8 sets**	65.9	85.5	86.6	81.95	67.5	80.9	87.1	76.8
Random **5 models/set in 8 sets**	61.85	65.9	65.9	55.67	68.55	63.4	56.7	68.0
AllPartitions **5 models/set in 8 sets**	78.3	84.53	87.6	81.44	72.68	86.0	84	79.9

of 200 random models to create 8 sets of random models for comparison against the AllPartitions sets. The factor levels for random models as shown in Table 4. The levels range from very small to large levels covering a larger portion of the input domain in terms of model size allowing us to compare model fragments based test models against random test models of varying sizes.

To summarize, we generate 360 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. The total computation time for the experiments which includes model generation and mutation analysis is about 50 hours. We discuss the results of mutation analysis in the following section.

5.3 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 random and model fragments based strategies. For instance, the lowest mutation score using AllRanges is 65.9 %. This corresponds to set 1 where the factor levels are 1,5,5,25,4,5 (see Column for set 1 in Table 3) and highest mutation scores are 86.6 and 87.1% 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 3).
- We observe a strong correlation of the mutation score with the number of Class and Association objects due to the nature of the injected mutation operators. The creational, navigational, and filtering mutation operators injected in the model transformation are killed by input test models using a large number of Class and Association objects. However, we see that random models with both large and small number of Class and Association objects are not able to have a mutation score above 72%. There is a clear need for more knowledge to improve this mutation score.
- We observe that AllPartitions test sets containing only 5 models/set gives a score of maximum 87.1%. The AllPartitions strategy provides useful knowledge to improve efficiency of test models.

We random test sets with model fragment guided sets in the *box-whisker* diagram shown in Figure 4. 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

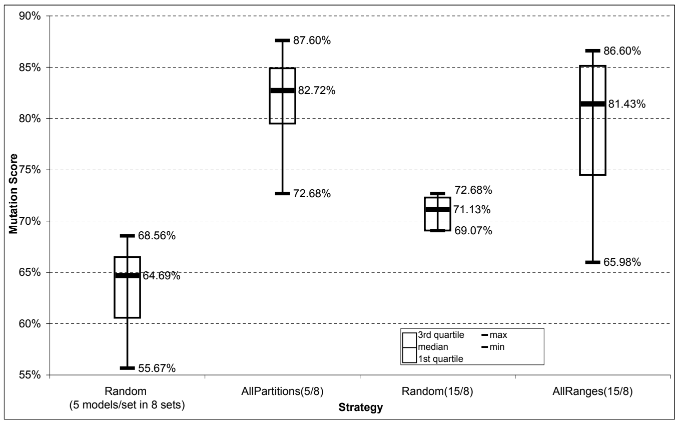


Fig. 4. Box-whisker Diagram to Compare Automatic Model Generation Strategies

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 4 we shown 4 boxes with whiskers for random 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 random sets although the random sets were selected using models of larger size.
– The high median mutation scores for strategies AllRanges 81% and AllPartitions 82.7% indicate that both these strategies return consistently good test sets. The median for AllPartitions 82.72% is highest among all 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 5 models/set using AllPartitions gives mutations scores equal or greater than 15 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* [17] with 5 models compared to 15 models.

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 random 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 randomly generated models do not contain inheritance relationships as it is not imposed by the meta-model.

What about the 12% of the mutants that remain alive given that the highest mutation score is 87.6%? We note by an analysis of the live mutants that they are the same for both AllRanges and AllPartitions. There remain 25 live mutants in a total of 200 injected mutants (with 6 equivalent mutants). In the median case the AllRanges strategy gives a mutation score of 81.43% and while AllPartitions gives a mutation score of 82.73%. 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 large 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. Further, 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.

We also neglect the effect of the constraint solver which is Alloy on the variation of the mutation scores. Relatively small boxes in the box-whisker diagram would be ideal to ascertain the benefits of test generation strategies. This again requires the generation of several thousand large and small models including multiple solutions for the same input specification. This will allow us to statistically minimize the external effects caused by Alloy and Boolean SAT solver allowing us to correctly qualify only the input generation strategies.

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) [18] [19] [20]. To extend such test criteria to MDE we have presented in [8] 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 [8].

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. [21]. 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 [22] 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 [23] 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 [24]. Generated models in both these approaches do not satisfy the constraints on the meta-model. In [25] 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 [4] 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 systematically generate several hundred models driven by knowledge/constraints of model fragments [8]. Statistically relevant test model sets are generated from a factorial experimental design [16] [26].

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 [9]. In previous work [10] 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 [2]. This transformation [12] was proposed in the MTIP Workshop in MoDeLs 2005 as a comprehensive and representative case study to evaluate model transformation languages.

7 Conclusion

Black-box testing exhibits the challenging problem of developing efficient model generation strategies. In this paper we present Cartier, a tool to generate hundreds of models conforming to the input domain and guided by different strategies. We use these test sets to compare four strategies for model generation. All test sets using these strategies detect faults given by their mutation scores. We generate test sets using only the input meta-model. The comparison partitioning strategies with unguided generation taught us that both strategies AllPartitions and AllRanges look very promising. Partitioning strategies give a maximum mutation score of 87% compared to a maximum mutation score of 72% in the case of random test sets. 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 efficiency 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.

References

1. 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)
2. 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
3. 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)

4. 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)
5. Budinsky, F.: Eclipse Modeling Framework. The Eclipse Series. Addison-Wesley (2004)
6. OMG: The Object Constraint Language Specification 2.0, OMG: ad/03-01-07 (2007)
7. Mahajan, Y.S., Z. Fu, S.M.: Zchaff2004: An efficient sat solver. In: Lecture Notes in Computer Science SAT 2004 Special Volume LNCS 3542. (2004) 360–375
8. Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.L.: Towards dependable model transformations: Qualifying input test data. Software and Systems Modelling (Accepted) (2007)
9. DeMillo, R., R.L., Sayward, F.: Hints on test data selection : Help for the practicing programmer. IEEE Computer **11**(4) (1978) 34 – 41
10. Mottu, J.M., Baudry, B., Traon, Y.L.: Mutation analysis testing for model transformations. In: Proceedings of ECMDA'06, Bilbao, Spain (July 2006)
11. Vaziri, M., Jackson, D.: Some shortcomings of ocl, the object constraint language of uml. In: TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00), Washington, DC, USA, IEEE Computer Society (2000) 555
12. 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)
13. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Tools and Algorithms for Construction and Analysis of Systems, Braga,Portugal (March 2007)
14. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: Uml2alloy: A challenging model transformation. In: MoDELS. (2007) 436–450
15. Jackson, D.: http://alloy.mit.edu. (2008)
16. Pfleeger, S.L.: Experimental design and analysis in software engineering. Annals of Software Engineering (2005) 219–253
17. 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)
18. 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)
19. 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
20. Gutjahr, W.J.: Partition testing versus random testing: the influence of uncertainty. IEEE TSE **25** (1999) 661–674
21. 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)
22. 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)
23. 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)
24. 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.
25. 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)
26. Federer, W.T.: Experimental Design: Theory and Applications. Macmillan (1955)

A Concise Version of Alloy Model Synthesized by Cartier

```
module tmp/simpleUMLCD
open util/boolean as Bool
sig ClassModel{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 the generated class diagram
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=>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=>c1=c2}
//There is exactly one ClassModel object
fact oneClassModel {#ClassModel=1}
/*All Classifier objects are contained in a ClassModel*/
fact classifierContainment {
all c:Classifier | c in ClassModel.classifier}
//All Association objects are contained in a ClassModel
fact associationContainment {
all a:Association| a in ClassModel.association}
/*A Classifier must have a unique name in the class diagram*/
fact uniqueClassifierName {
all c1:Classifier, c2:Classifier |c1.name==c2.name => 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 => (a1 = a2 or a1.src != a2.src)}
/*Model Transformation Pre-condition*/
fact atleastOnePrimaryAttribute {
all c:Class| one a:c.attrs | a.is_primary==True}
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}
fact noAttribAndAssocSameName{all c:Class,assoc:Association |
all a:c.attrs|(assoc.src==c)=>a.name!=assoc.name}
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}
```