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

Automatic Model Synthesis to Test Transformations

Sagar Sen, Jean-Marie Mottu, Benoit Baudry

INRIA Rennes - Bretagne Atlantique / IRISA, Université Rennes 1, Triskell Team, Campus de Beaulieu, 35042 Rennes Cedex, France

e-mail: {ssen, bbaudry}@irisa.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

Received: date / Revised version: date

Abstract Testing a *model transformation* requires input test models which are graphs of inter-connected objects. These test models must conform to the transformation's *input meta-model* and to *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 conform to a possibly large input metamodel such as UML and must simultaneously conform to several meta-constraints. In this paper, we present the model generation tool CARTIER to automatically generate test models for any given model transformation. CARTIER first constructs a precise specification of a transformation's effective input domain via *meta-model pruning*. CARTIER transforms the effective input domain including knowledge from heterogenous sources to a common formal specification in the language ALLOY. CARTIER solves the ALLOY model to generate test models that are guided by a test strategy. These test models often help discover new pre-conditions for the transformation. We re-generate test models once we discover all pre-condition constraints. We qualify test models generated using input domain coverage strategies via *mutation analysis*. We show that sets of test models satisfying coverage strategies give mutation scores of up to 93% vs. 70% in the case of unguided/random generation. These scores are based on analysis of 3200 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 the validation of transformations remains a hard problem [1]. In this paper, we address the challenges in validating model transformations via *black-box automatic test data generation*. We think that black-box testing is an effective approach to validating transformations due to the diversity of transformation languages based on graph rewriting [2], imperative execution (Kermeta [3]), and rule-based transformation (ATL [4]) that render language specific formal methods and white-box testing currently 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, transformation pre-conditions, and test strategies. Manually specifying several hundred test models targeting various testing objectives is a tedious task and in many cases impossible since the modeller may have to simultaneously satisfy numerous possibly inter-related constraints.

In this paper, we present the tool and framework CARTIER for *automatic test model generation* based on the general idea of *constraint satisfaction* in the domain of models. CARTIER has to address two main problems for test generation: identify a precise model of the transformation's input domain; automatically select relevant test models in the input domain. The first issue is related to the fact that the input domain of a transformation is generally described with a general purpose metamodel (e.g., UML). However, the effective input domain, that captures only the set of models that can be transformed, is much smaller than the set of instances of the general purpose metamodel. CARTIER can prune the metamodel in order to explicitly build the sub part of the metamodel that the transformation can manipulate. CARTIER also assists the definition of pre conditions on the metamodel to make the input domain more precise. Once the input domain is precisely modelled, CARTIER can generate model from this domain. CARTIER either generate any model without guidance or it can use test strategies in order to have models that cover the input domain [5].

Are the test models generated by CARTIER able to detect bugs in a model transformation? We answer this question by generating and comparing sets of test models using different testing strategies. Specifically, we consider two testing strategies: *unguided* and *input domain coverage strategies* [5]. We use *mutation analysis* [6] [7] for model transformations to compare these testing strategies. Mutation analysis serves as a *test oracle* to determine the relatively adequacy of generated test sets.

We perform experiments to generate test models using different testing strategies and qualify them using mutation analysis. We generate test models for the representative model transformation of Unified Modelling Language Class Diagram (UMLCD) to Relational Database Management Systems (RDBMS) models called class2rdbms. The mutation scores show that input domain coverage strategies guide model generation with considerably higher bug detection abilities (93%) compared to unguided generation (70%). 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 generation can be attributed to the fact that coverage strategies enforce several aspects on test models that unguided generation fail 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 *scientific contribution* in this paper is based on a combination of several recently published ideas. In [8], the authors for the first time present CARTIER demonstrating the possibility of automatically generating a variety of test models. In [9], the authors generate hundreds of test models using different strategies to show that automatically generated test models using partitioning strategies can indeed detect bugs. We validate the bug detecting effectiveness of the generated test models using mutation analysis of model transformations [7]. However, three important questions remain:

– **Question 1:** How can we scale the approach to generating test models for large input meta-models such the UML?
– **Question 2:** Does the model transformation pre-condition precisely specify the input domain of a model transformation? If not, can automatically generated test models help improve the pre-condition by presenting unforeseen and unwanted modelling patterns?
– **Question 3:** Are we consistently able to generate effective test models for a given strategy using our approach?

The precise contributions of this paper address exactly these problems. We enlist them below:

– **Contribution 1:** We use the recently proposed *meta-model pruning* transformation [10] to prune a large input meta-model such as the UML to a subset called the effective input meta-model. The effective input meta-model contains only classes, properties, their dependencies relevant the transformation under test. The often smaller effective input meta-model is transformed to a small formal representation in ALLOY. In contrast, transforming a large input meta-model such as the whole of UML to ALLOY results in a formal model that renders SAT solving infeasible due to the large number of signatures or facts.
– **Contribution 2:** We show how automatically generated test models can help us improve a model transformation's pre-condition. For instance, the test models we generate for the case study transformation class2rdbms helps us discover new pre-condition constraints. These pre-conditions were not initially envisaged by the panel of world experts in model-driven engineering who propose the class2rdbms as the benchmark case study at the MTIP workshop [11]. We show that automatic generation can help us rapidly discover structures that human or even experts cannot preview in advance or require several years of transformation usage experience.
– **Contribution 3:** We show that CARTIER consistently generates effective test models for a given strategy. We illustrate consistency by demonstrating that generating multi-

ple test models for the same test strategy does not significantly change mutation scores. These test models correspond to multiple non-isomorphic solutions obtained using ALLOY's symmetry breaking scheme [12].

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 CARTIER. In Section 4, we describe the CARTIER methodology 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 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 (For example, 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 $MT_{Requirements}$.

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.

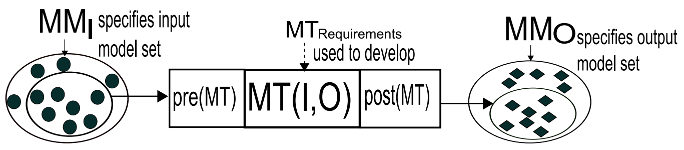

MMI specifies input model set
MTRequirements used to develop
MMO specifies output model set
pre(MT)
MT(I,O)
post(MT)

Fig. 1 A Model Transformation

2.1 Transformation Case Study

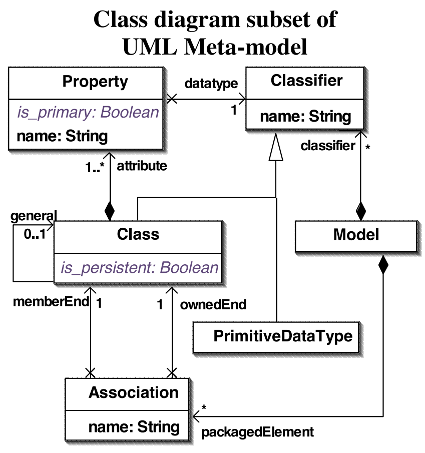
Our case study is the transformation from UML Class Diagram models to RDBMS models called class2rdbms. In this section we briefly describe class2rdbms and discuss why it is a representative transformation to validate test model generation strategies.

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 a subset of the UML input meta-model for class2rdbms. The concepts and relationships in the input meta-model are stored as an Ecore model [13] (Figure 2 (a)). The invariants on the UMLCD Ecore model, expressed in Object Constraint Language (OCL) [14], are shown in Figure 2 (b). The Ecore model and the invariants together represent the true 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 [15] 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 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 meta-model of UML class diagram model covers all major meta-modelling concepts such as inheritance, composition, finite and infinite multiplicities. The entire UML input meta-model serves as a large input meta-model to demonstrate meta-model pruning to an effective input meta-model containing only class diagram concepts.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 [7]) enabling us to test essential model transformation features. Among the limitations the UMLCD meta-model does not contain Integer and Float attributes. The number of classes in the 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. However, this not really limitation in our approach as we claim that specifying a test model requires only a small subset of the entire meta-model and extracting this subset via meta-model pruning is part of our methodology.

Model generation is relatively fast but performing mutation analysis is extremely 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



(a)

Samples of **OCL Invariants**

```
context Class
    inv noCyclicInheritance:
        not self.allGenerals()->includes(self)
    inv uniqueAttributesName:
        self.attribute->forAll(att1, att2 |
            att1.name=att2.name implies att1=att2)

context Model
    inv uniqueClassifierNames:
        self.classifier->forAll(c1, c2 |
            c1.name=c2.name implies c1=c2)
    inv uniqueClassAssociationSourceName :
        self.packagedElement->forAll(ass1, ass2 |
            ass1.name=ass2.name implies
            (ass1=ass2 or ass1.src != ass2.src))
```

(b)

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

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 Foundations

This section presents the foundations required to explain the CARTIER methodology for automatic test model generation and validation 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 including CARTIER and meta-model pruning. We describe Kermeta's implementation of model typing in Section 3.2 which helps us perform all type conformance operations in our approach. In Section 3.3, we present the meta-model pruning algorithm to obtain the effective input meta-model or the true input domain of a model transformation. We briefly describe CARTIER's transformation to ALLOY and the automatic model generation mechanism in Section 3.4. Model generation in this paper is driven by coverage criteria based testing strategies. These testing strategies are described in Section 3.5. Finally, the bug detecting effectiveness of test models generated using different testing strategies is done by mutation analysis. Mutation analysis for model transformations is described in Section 3.6.

3.1 Kermeta

Kermeta is a language for specifying meta-models, models, and model transformations that are compliant to the Meta Object Facility (MOF) standard [16]. The object-oriented meta-language MOF supports the definition of meta-models 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 meta-models [17]. 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. We use Kermeta to implement all model transformations in this paper.

3.2 Model Typing

The last version of the Kermeta language integrates the notion of model typing [18], which corresponds to a simple extension to object-oriented typing in a model-oriented context. Model typing can be related to structural typing found in languages such as Scala. Indeed, a model typing is a strategy for typing models as collections of interconnected objects while preserving type conformance, used as a criterion of substitutability.

The notion of model type conformance (or substitutability) has been adapted and extended to model types based on Bruce's notion of type group matching [19]. The matching relation, denoted <#, between two meta-models defines a function of the set of classes they contain according to the following definition:

> Meta-model *M'* matches another meta-model *M* (denoted *M' <# M*) iff for each class *C* in *M*, there is one and only one corresponding class or subclass *C'* in *M'* such that every property *p* and operation *op* in *M.C* matches in *M'.C'* respectively with a property *p'* and an operation *op'* with parameters of the same type as in *M.C*.

This definition is adapted from [18] and improved here by relaxing two strong constraints. First, the constraint related to the name-dependent conformance on properties and operations was relaxing by enabling their renaming. The second constraint related to the strict structural conformance was relaxing by extending the matching to subclasses.

Let's illustrate model typing with two meta-models *M* and *M'* given in Figures 3 and 4. These two meta-models have model elements that have different names and the meta-model *M'* has additional elements compared to the meta-model *M*.

> *C1 <# COne* because for each property *COne.p* of type *D* (namely, *COne.name* and *COne.aCTwo*), there is a matching property *C1.q* of type *D'* (namely, *C1.id* and *C1.aC2*), such that *D' <# D*.
>
> Thus, *C1 <# COne* requires *D' <# D*, which is true because:
>
> – *COne.name* and *C1.id* are both of type *String*.
> – *COne.aCTwo* is of type *CTwo* and *C1.aC2* is of type *C2*, so *C1 <# COne* requires *C2 <# CTwo* or that a subclass of *C2* matches *CTwo*. Only *C3 <# CTwo* is true because *CTwo.element* and *C3.elem* are both of type *String*.

Thus, matching between classes may depend on the matching of their related dependent classes. As a consequence, the dependencies involved when evaluating model type matching are heavily cyclical [20]. The interested reader can find in [20] the details of matching rules used for model types.

3.3 Meta-model Pruning

Meta-model pruning [10] is an algorithm that outputs an effective subset meta-model of a possible large input meta-model such as the UML. The output effective meta-model conserves a set of required types and properties (given as input to meta-model pruning) and all its obligatory dependencies (computed by the algorithm). The algorithm prunes every other type and property. In the type-theoretic sense the resulting effective meta-model is a *supertype* of the large input meta-model. We verify the supertype property using model typing [18]. We concisely describe the meta-model pruning algorithm in the following paragraphs.

Given a possibly large meta-model such as UML that may represent the input domain of a model transformation we ask the question : Does the model transformation process models containing objects of all possible types in the input meta-model? In several cases the answer to this question may be no. For instance, a transformation that refactors UML models only processes objects with types that come from concepts in the UML class diagrams subset but not UML Activity, UML Statechart, or UML Use case. How do we obtain this effective subset? This is the problem that meta-model pruning solves.

The principle behind pruning is to preserve a set of required types T_{req} and and required properties P_{req} and prune away the rest in a meta-model. The authors of [10] present a set of rules that help determine a set of required types T_{req} and required properties P_{req} given a meta-model MM and an initial set of required types and properties. The initial set may come from various sources such as manual specification or a *static analysis of a model transformation* to reveal used types. A rule in the set for example adds all super classes of a required class into T_{req}. Similarly, if a class is in T_{req} or is a required class then for each of its properties p, add p into P_{req} if the lower bound for its multiplicity is > 0. Apart from rules the algorithm contains options which allow better control of the algorithm. For example, if a class is in T_{req} then we add all its sub-classes into T_{req}. This optional rule is not obligatory but may be applicable under certain circumstances giving the user some freedom. The rules are executed where the conditions match until no rule can be executed any longer. The algorithm terminates for a finite meta-model because the rules do not remove anything from the sets T_{req} and P_{req}.

Once we compute the sets T_{req} and P_{req} the algorithm simply removes the remaining types and properties to output the effective meta-model MM_e. The effective meta-model MM_e generated using the algorithm in [10] has some very interesting characteristics. Using model typing (discussed in Section 3.2) we verify that MM_e is a *supertype* of the meta-model MM. This implies that all operations written for MM_e are valid for the large meta-model MM.

In CARTIER, we use meta-model pruning to prune the input meta-model MM_I to obtain the effective input meta-model eMM_I. The effective input meta-model eMM_I contains all the classes and properties used in the transformation un-

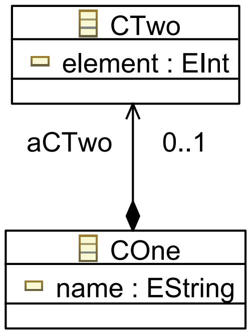


Fig. 3 Metamodel *M*.

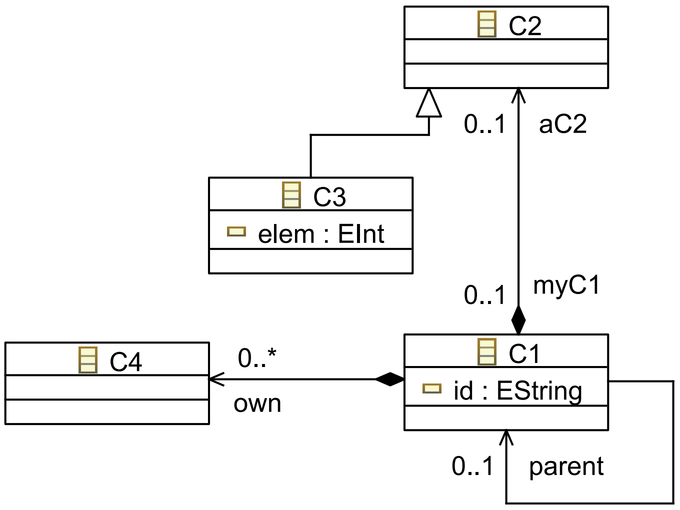


Fig. 4 Metamodel *M'*.

der test and their obligatory dependencies. For our case study transformation class2rdbms we prune UML with a required set of types namely Class, Classifier, Association, Property, PrimitiveDataType and the top-level class Model to obtain the effective input meta-model previously shown in Figure 2. The pruned meta-model UMLCD is a subset of UML and a its *supertype*. The supertype property implies that any transformation written for UMLCD is backward compatible with UML. All instances of UMLCD are instances of UML. This result allows us to exclusively transform UMLCD to ALLOY and not the whole of UML.

3.4 CARTIER *Transformation to* ALLOY *and Automatic Model Generation*

We use the tool CARTIER previously introduced in our paper [8] 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 [21], launch a SAT solver such as MiniSAT [22] or ZChaff [23] to generate models that conform to the input domain of a model transformation.

CARTIER transforms a model transformation's effective input meta-model (obtained via meta-model pruning described in Section 3.3) expressed in the Eclipse Modelling Framework [13] format called Ecore using the transformation rules presented in [8]. Basically, classes in the effective input meta-model are transformed to ALLOY signatures and implicit constraints such as inheritance, opposite properties, and multiplicity constraints are transformed to ALLOY facts. The 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 [24]. 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 fact in Listing 1.

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

Listing 1 ALLOY Fact for No Cyclic Inheritance

The generated ALLOY model for the UMLCD meta-model is given in Appendix A. This ALLOY model only describes the effective input domain of the transformation. Generating model instances of the ALLOY model 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 Section 3.5 we illustrate how one can guide model generation using strategies based on input domain partitioning.

3.5 Test Strategies

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 4. 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 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 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 [5]. The accompanying tool called Meta-model Coverage Checker (MMCC) [5] 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 [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 *mfAllPartitions1* chosen using AllPartitions defines both *partition1* and *partition2*.

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

```
pred mfAllRanges7
{
  some c : Class | #c.attribute=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 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 [25] 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. 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 (see Listing 4).

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 $c \mid \#c.general = 0$
mfAllRanges6	A Class $c \mid \#c.general = 1$
mfAllRanges7	A Class $c \mid \#c.attribute = 1$
mfAllRanges8	A Class $c \mid \#c.attribute > 1$
mfAllRanges9	An Property $a \mid a.is\_primary = True$
mfAllRanges10	An Property $a \mid a.name =$“”
mfAllRanges11	An Property $a \mid a.name! =$“”
mfAllRanges12	An Property $a \mid \#a.datatype = 1$
mfAllRanges13	An Association $as \mid as.name =$“”
mfAllRanges14	An Association $as \mid \#as.memberEnd = 0$
mfAllRanges15	An Association $as \mid \#as.memberEnd = 1$
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 $c1, c2 \mid \#c1.general = 0$ and $\#c2.general = 1$
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 mfAllRange7a
{
  some c:Class | #c.attribute = 0
}

run mfAllRanges7 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
{ ...
  attribute : some Property
  ...
}
```

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 1.

3.6 Qualifying Models: Mutation Analysis for Model Transformation Testing

We generate sets of test models using different strategies and qualify these sets via mutation analysis [6]. 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 [7]. 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 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 [7]. We have clearly identified faults and equivalent mutants to study the effect of our generated test models.

4 Automatic Test Model Generation and Qualification Methodology

We outline the methodology for test generation using CARTIER and qualification of the generated test models via mutation analysis in Figure 5. The methodology encapsulates all ideas we present in Section 3 into a workflow. CARTIER is based on technologies like Kermeta, Eclipse Modelling Framework (Ecore), and ALLOY as shown in the figure. Concisely, the test model generation workflow follows the steps:

1. CARTIER performs static analysis on the model transformation MT to obtain the initial set of used types and properties.
2. CARTIER performs meta-model pruning of MM_I using these used types and properties to obtain the effective input meta-model eMM_I (details in Section 3.3)
3. CARTIER transforms eMM_I, its invariants, the transformation pre-condition $pre(MT)$ and test strategy to an ALLOY model (details in Sections 3.4, 3.5)
4. CARTIER generates models to detect inconsistencies in test strategy predicates. It eliminates those that are inconsistent with eMM_I and $pre(MT)$ (details in Section 3.5)
5. Finally, CARTIER generates sets of test models that satisfy all consistent predicates representing test strategies in a finite scope using run commands for each predicate (details in Section 3.5). It can also generate multiple non-isomorphic test models by soliciting ALLOY's symmetry breaking scheme [12] currently applicable to the MiniSAT [22] SAT solver.

The generated models may lead to raising of exceptions in the model transformation MT as its initial pre-condition definition may not have been well defined. In the following Section 4.1 we show how automatically generated models resulted in discovery of patterns that were not foreseen by experts who original designed the transformation class2rdbms.

After discovering pre-conditions that no longer lead to generation of models that are raise exceptions we regenerate sets of test models. We qualify the sets of generated test models via mutation analysis (see Section 3.6).

4.1 Pre-condition Improvement

The execution of a transformation helps us discover new constraints for the pre-condition $pre(MT)$ of the transformation MT. In this sub-section we illustrate how some of the con-

Table 2 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

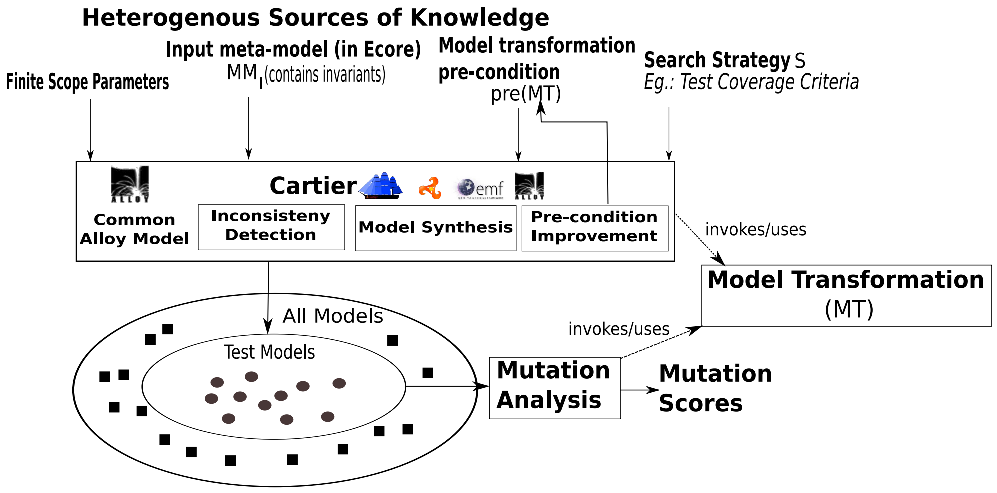

Heterogenous Sources of Knowledge
Input meta-model (in Ecore)
MM$_I$(contains invariants)
Model transformation pre-condition
pre(MT)
Search Strategy S
Eg.: Test Coverage Criteria
Finite Scope Parameters
Cartier
Common Alloy Model
Inconsisteny Detection
Model Synthesis
Pre-condition Improvement
invokes/uses
Model Transformation (MT)
All Models
Test Models
invokes/uses
Mutation Analysis
Mutation Scores

Fig. 5 CARTIER Methodology for Automatic Test Generation and Mutation Analysis based Qualification

straints in the pre-condition of the transformation class2rdbms are discovered.

The discovery of a pre-condition starts with the detection of a fault during the execution of automatically generated models. The exception handling mechanism in Kermeta allows us to detect and catch these exceptions. First, we prevent the lock of the execution when a transformation runs into infinite loop. 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.

Second, we detect more complex inconsistencies when output models produced from an automatically generated input model are not in the 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 6 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).

Our tool isolates inconsistent output models and corresponding input models. We then use a traceability mechanism and tool such as in [26] to restrain the analysis of these models on excerpts such as the one illustrated in Figure 6. Class

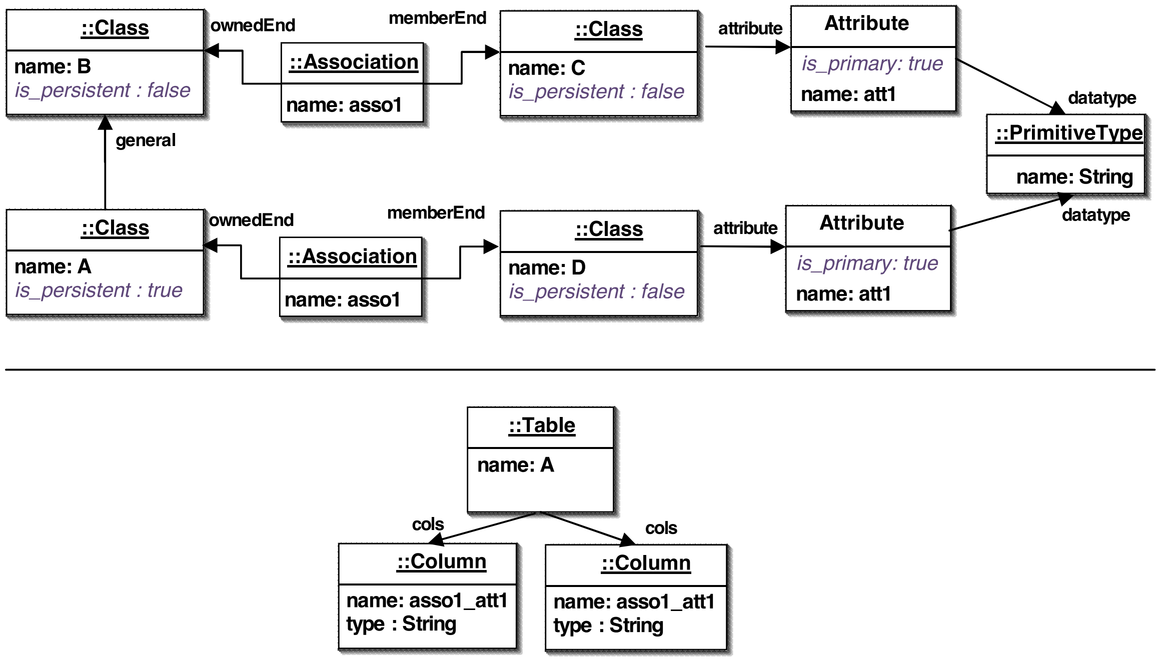


Fig. 6 Model Excerpt for Pre-condition Improvement

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. 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 6 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 3.5 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 [27]. We consider the *exact number of objects for each class* in the effective input meta-model as factors for 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 3. 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 1). 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 3 (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 3. 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.

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 3.

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.

Table 4 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

Table 3 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	

The computation time for generating 3200 models was about 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 4 (row 2). Mutation scores for test sets obtained using AllPartitions are shown in Table 4 (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 3) 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 3).
– 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 com-

bine 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 7. 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 7 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* [28] 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 strat-

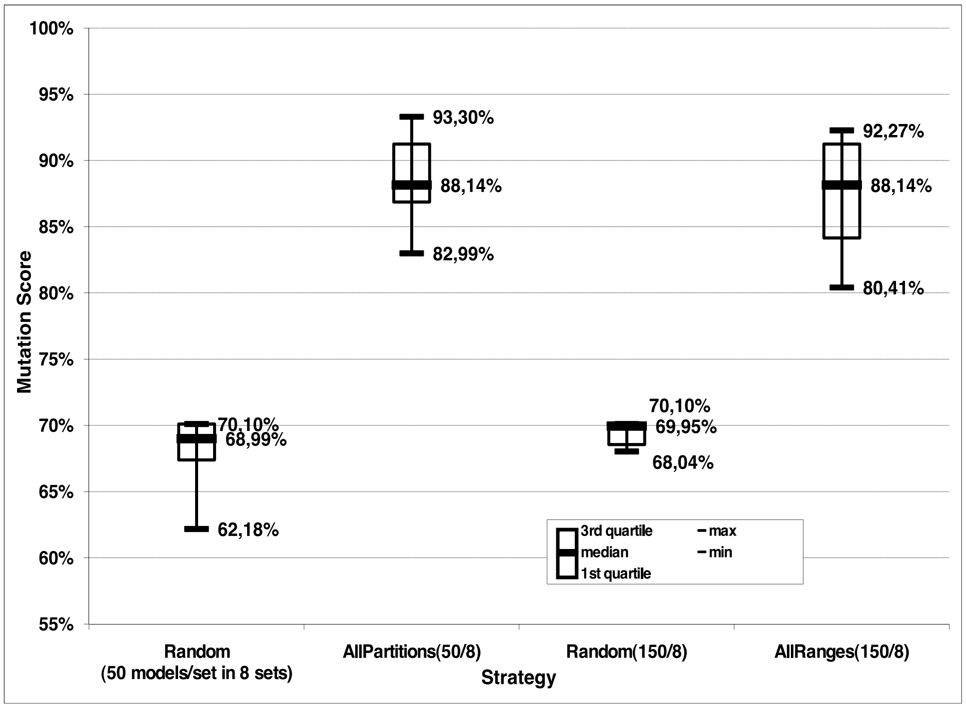


Fig. 7 Box-whisker Diagram to Compare Automatic Model Generation Strategies

egy 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 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) [29] [30] [31]. 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. [32]. 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 [33] 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 [34] 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 [35]. Generated models in both these approaches do not satisfy the constraints on the meta-model. In [36] 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 [8] 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 [5]. Statistically relevant test model sets are generated from a factorial experimental design [27] [37].

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 [6]. In previous work [7] 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

Black-box testing exhibits the challenging problem of developing efficient model generation strategies. In this paper we present CARTIER, a tool to generate models conforming to the input domain and guided by different test strategies. First, CARTIER helps us precisely specify the input domain of a model transformation via meta-model pruning and precondition improvement. Second, we use CARTIER 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 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.

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,
  general : lone Class,
  attribute : some Property

}

sig Association

{
  name: Int,
  memberEnd: one Class,
  ownedEnd: one Class

}

sig Property
{
  name: Int,
  is_primary : Bool,
  datatype: one Classifier
}

//Meta-model constraints

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

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

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

fact uniquePropertyNames
```

```
{
  all c:Class | all a1:  c.attribute , a2: c.attribute | 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.attribute , a2 : c2.attribute | 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 atleastOnePrimaryProperty
{
  all c:Class | one a:c.attribute | a.is_primary =True
}

fact no4CyclicClassProperty
{
  all a:Property | a.datatype in Class implies all a1:a.datatype.attribute | a1.
        datatype in
  Class implies all a2:a.datatype.attribute | a2.datatype in Class implies all a3
        :a.datatype.attribute|a3.datatype
   in Class implies all a4:a.datatype.attribute | a4.datatype in
         PrimitiveDataType
}

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

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

fact no2CycleNonPersistent
{
  all a1: Association, a2:Association |
  (a1.memberEnd = a2.ownedEnd and a2.memberEnd = a1.src) implies
  a1.ownedEnd.is_persistent = True or a2.ownedEnd.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 no4CyclicClassProperty{
  all a:Property | a.datatype in Class => all a1:a.datatype.attribute|a1.datatype
         in Class => all a2:a.datatype.attribute|a2.datatype in Class => all a3:
        a.datatype.attribute|a3.datatype in Class => all a4:a.datatype.attribute
        |a4.datatype 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.attribute | (assoc.ownedEnd == c) =>
          a.name != assoc.name
}


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

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



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



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

fact assocPersistentClass
{
  all a:Association | a.ownedEnd.is_persistent=True implies a.memberEnd not in a.
         ownedEnd.^general
}



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

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


 }


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


fact classCantTypeOfAllofItsProperty
{
 all c:Class | all a: (c.attribute+c.^general.attribute) | a.datatype !=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.^general | B.is_persistent == True implies (no a1:
         Association, a2:Association |
(a1.ownedEnd = A and a2.ownedEnd=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.^general | B.is_persistent == True implies (no a1: A.
         attribute, a2:B.attribute |
(a1.name=a2.name))
}

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

fact noAssocBetweenClassInHierarchy
{
  all a : Association | all c: Class | (a.ownedEnd =c implies a.memberEnd not in
         c.^general) and (a.memberEnd =c implies a.ownedEnd not in c.^general)
}
```

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 Con-

ference 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. DeMillo, R., R.L., Sayward, F.: Hints on test data selection : Help for the practicing programmer. IEEE Computer **11**(4) (1978) 34 – 41
7. Mottu, J.M., Baudry, B., Traon, Y.L.: Mutation analysis testing for model transformations. In: Proceedings of ECMDA'06, Bilbao, Spain (July 2006)
8. 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)
9. Sen, S., Baudry, B., Mottu, J.M.: Automatic model generation strategies for model transformation testing. In: ICMT. (2009) 148–164
10. 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)
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. Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. In: SAT 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing. Volume 155., June (2001) 1539–1548
13. Budinsky, F.: Eclipse Modeling Framework. The Eclipse Series. Addison-Wesley (2004)
14. OMG: The Object Constraint Language Specification 2.0, OMG Document: ad/03-01-07 (2007)
15. 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
16. OMG: Mof 2.0 core specification. Technical Report formal/06-01-01, OMG (April 2006) OMG Available Specification.
17. 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
18. Steel, J., Jézéquel, J.M.: On model typing. Journal of Software and Systems Modeling (SoSyM) **6**(4) (December 2007) 401–414
19. Bruce, K.B., Vanderwaart, J.: Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. Electronic Notes in Theoretical Computer Science **20** (1999) 50–75
20. Steel, J.: Typage de modèles. PhD thesis, Université de Rennes 1 (April 2007)
21. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Tools and Algorithms for Construction and Analysis of Systems, Braga,Portugal (March 2007)
22. Een, N., Srensson, N.: Minisat: A sat solver with conflict clause minimization. In: SAT. (2005)

23. 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
24. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: Uml2alloy: A challenging model transformation. In: MoDELS. (2007) 436–450
25. Jackson, D.: http://alloy.mit.edu. (2008)
26. 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)
27. Pfleeger, S.L.: Experimental design and analysis in software engineering. Annals of Software Engineering (2005) 219–253
28. 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)
29. 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)
30. 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
31. Gutjahr, W.J.: Partition testing versus random testing: the influence of uncertainty. IEEE TSE **25** (1999) 661–674
32. 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)
33. 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)
34. 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)
35. 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.
36. 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)
37. Federer, W.T.: Experimental Design: Theory and Applications. Macmillan (1955)