On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing

Sagar Sen, Benoit Baudry, Jean-Marie Mottu
IRISA/INRIA
Campus universitaire de Beaulieu, 35000 Rennes, France
ssen,bbaudry,jmottu@irisa.fr

Abstract

Testing remains a major challenge for model transformation development. Test models that are used as test data for model transformations, are constrained by various sources of knowledge that is expressed in different formalisms. Thus, in order to automatically generate test models it is necessary to interpret these different sources of knowledge and combine them into a consistent set of information that can be used for model synthesis. In this paper, we identify sources of testing knowledge and present our tool Cartier *that uses* Alloy *as the first-order relational logic language to represent combined knowledge in the form of constraints. The constraints are solved leading to a selection of qualified test models from the input domain of a model transformation. We illustrate our approach using the* Unified Modeling Language Class Diagram to Relational Database Management Systems *transformation as a running example.*

1. Introduction

Model Driven Engineering(MDE) is grounded on the idea of representing models of software systems at different levels of abstraction using various modelling languages. Programs that automatically manipulate models are called *model transformations*. These transformations can automate important steps in a development process such as refinement towards a more concrete model, re-factoring to improve maintainability or readability of the design, etc. Thus it is crucial to develop efficient techniques to validate the transformations so that they are robust enough to handle the processing of a variety of models in their lifetime of use.

In this paper we are interested in testing model transformations in order to validate them. Testing of model transformations consists in synthesising a large number of different input models, running the program and verifying the result. In this work, we focus on the automatic synthesis of input models for testing. We call these models test models.

Automatic synthesis of test models is a difficult task due to large amount of constraints that these models have to satisfy. There are two kinds of constraints that must be considered for test models: the constraints that define the licit input models for the transformation and the constraints that aim at selecting models with a specific testing goal. In addition to this large set of constraints, another challenge consists in dealing with the heterogeneous formalisms in which these constraints are expressed.

The solution studied in this work focuses on four types of constraints expressed in different formalisms. Two types of constraints define the set of licit models for the transformation: the metamodel and the pre condition. The metamodel is specified in two parts: a structure built with the Ecore language, and constraints on this structure expressed in OCL. The pre-condition for the transformation is also expressed in OCL. Two types of constraints are used to select test models among the whole set of licit models: partitions on the input domain and test model objectives that are derived from the requirements of the transformation. The partitions are derived from the metamodel and are composed in model fragments according to the test criteria defined in [10]. These fragments are expressed in a dedicated language. Currently, there exists no particular modelling language to specify the requirements for a transformation or to express test model knowledge. Thus, we model them directly in Alloy, which is the underlying model language for test model synthesis.

In this paper, we present a tool Cartier that aims to combine all these knowledge from multiple formalisms to guide the automatic selection of test models from the set of all input models. The tool transforms all knowledge to a common constraint language. The common constraint language is first-order relational logic implemented in Alloy [13]. The relational logic program is transformed to a boolean satisfiability problem and solved using a SAT solver to generate (select) test models.

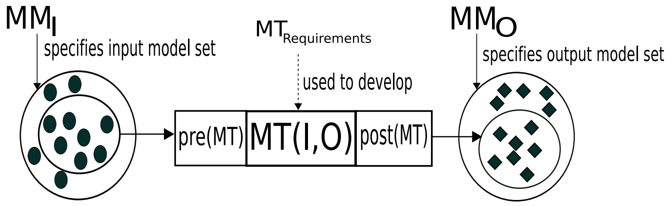


Figure 1. A Model Transformation

The paper is organized as follows. In Section 2 we present the details with examples of different sources of knowledge that are necessary for generating test models. All details and examples are based on the running example transformation of Unified Modeling Language Class Diagram (UMLCD) to Relational Database Management Systems (RDBMS), UMLCD_2_RDBMS which is also introduced in Section 2. The combination of these knowledge sources to a common constraint language is discussed in Section 3 where we present the Cartier software environment. In Section 4 we perform test model selection and show how our method is able to generate models that qualify a model transformation based on input domain coverage, pre-condition satisfaction and requirements coverage. Related work is given in Section 5. We conclude in Section 6. We provide, in Appendix A, the completely transformed Alloy model for direct execution on the Alloy analyzer.

2. Multi-formalism Knowledge for Testing Model Transformations

We start this section with a brief description of a model transformation. A model transformation $MT(I,O)$ is a program on a set of input models I to give 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. The set of all output models is specified by a meta-model MM_O. The pre-condition of the model transformation $pre(MT)$ constrains the set of all input models to its subset. 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 $MT_{Requirements}$.

Test model selection involves finding valid input models we call *test models* from the set of all input models I. These test models are valid in the sense that they belong to the input domain of the transformation (conform to the input meta-model). In addition to this, they are selected in order to satisfy other 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 $MT(I,O)$.

In order to select test models, we use knowledge from various sources and expressed in multiple formalisms. These formalisms are discussed in the following sub-sections. The running example we use is that of transformation from Unified Modeling Language Class Diagram to Relational Database Management Systems. This is a benchmark transformation proposed in the MTIP workshop at the MoDELS 2005 conference [5]. From a utility point of view there are several tools that serialize class diagram software designs to databases for storing and querying code designs with efficiency. An example of such a tool is Hibernate 3.0.

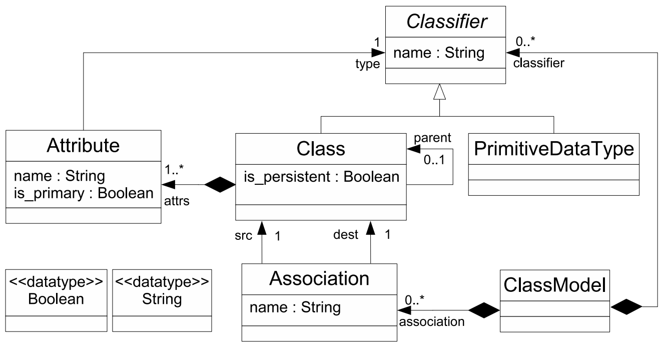


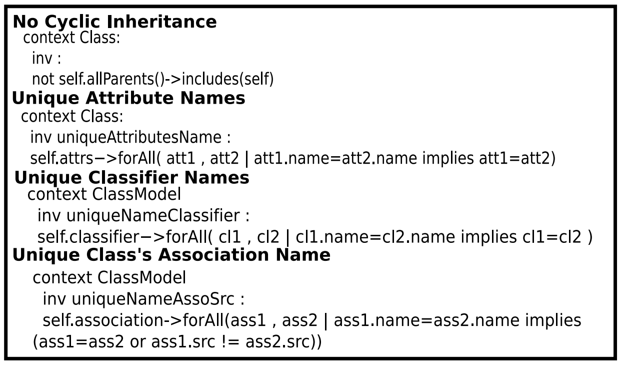
Figure 2. Simple Unified Modeling Language Class Diagram Ecore Model

2.1 Input Meta-model

The input domain or the set of all possible inputs to a model transformation is specified using a meta-model MM_I. For instance, the set of all input models for the transformation UMLCD_2_RDBMS is specified using a meta-model which is composed of an Ecore model in Figure 2 and a set of constraints in Object Constraint Language (OCL)[18] shown in Figure 3. The OCL constraints are expressed on the Ecore model. Ecore [12] is the Eclipse implementation of Meta-object Facility (MOF) [17] which is an Object Management Group (OMG) standard for developing a part of the meta-model. In the following sub-sections we will describe the Ecore model and the OCL constraints on the Ecore model for the UMLCD input domain for the UMLCD_2_RDBMS model transformation.

2.1.1 Ecore Model

The Ecore model part of the meta-model consists of *classes* with *properties*. For instance, in the UMLCD meta-model 2 we have the classes Class, ClassModel, Classifier, Association, and PrimitiveDataType. Each of these classes have properties. A property can be an *attribute* or a *reference* to another class.



No Cyclic Inheritance
```
 context Class:
   inv :
   not self.allParents()->includes(self)
```
Unique Attribute Names
```
 context Class:
   inv uniqueAttributesName :
   self.attrs−>forAll( att1 , att2 | att1.name=att2.name implies att1=att2)
```
Unique Classifier Names
```
 context ClassModel
   inv uniqueNameClassifier :
   self.classifier−>forAll( cl1 , cl2 | cl1.name=cl2.name implies cl1=cl2 )
```
Unique Class's Association Name
```
   context ClassModel
    inv uniqueNameAssoSrc :
    self.association->forAll(ass1 , ass2 | ass1.name=ass2.name implies
  (ass1=ass2 or ass1.src != ass2.src))
```

Figure 3. Important OCL Constraints on UML Class Diagram Ecore model

An attribute has a primitive type which is either Boolean, Integer, Float, or String. For instance, the class Class has an attribute is_persistent which is of type Boolean. It also has a reference attrs which refers to a collection of Attribute objects. The number of references is constrained by a *multiplicity/cardinality*. For instance, a Class object can be associated to $0..*$ (implying 0 to *many*) Attribute objects.

Another feature of a meta-model is a *containment*. For instance, Attribute objects that are referenced by a Class objects are contained in it. This is shown using a black diamond link in the class diagram. This imposes the constraint that an Attribute object can be contained in only one Class object.

A class can inherit properties from parent classes. For instance, the Class class inherits from the Classifier class. Hence, a Class inherits the property name of type String from Classifier. The basic data-types used in the Ecore model which are Boolean and String are obtained from the Java library of basic type definitions.

2.1.2 OCL Constraints

The OCL constraints on the meta-model are shown in Figure 3. OCL constraints are written when constraints can no longer be expressed using the language of Ecore.

For instance, the No Cyclic Inheritance constraint in the meta-model states that the set of all parents of a Class object cannot contain itself. Or in other words a Class cannot inherit its own properties. Such a constraint cannot be expressed using only an Ecore model.

2.2 Model Transformation Pre-condition

The input meta-model first specifies the set of all input models. However, the model transformation itself may have some constraints that need to be satisfied for it to correctly process an input model. This constraint is called a pre-condition. In this work, we consider pre conditions expressed as OCL constraints.

In the case of UMLCD_2_RDBMS transformation we have a pre-condition that states that every class must have at least one attribute with *is_primary* = *true*. This is necessary for storage and indexing in the resulting RDBMS model of the UMLCD.

The pre-condition in OCL is expressed as follows:

```
context Class:
inv atleastOnePrimaryAttribute:
self.attrs->select(att1|att1.is_primary)
->size()>=1
```

2.3 Test Model Objectives

A model transformation is developed by an engineer with respect to a set of requirements. The tester has the intent to check that these requirements are satisfied by the model transformation. Testers express a set of *tester's intents* which are in correspondence with the requirements of the model transformation.

The tester's intents are used to develop *test model objectives* which are expressed as objects with specific properties that must be present in the test model or as OCL constraints. Each such objective is geared to test a particular requirement of the model transformation. The *tester's intent* makes the relation between the test model objectives and the targeted requirement in the model transformation. There is no specific language at the moment to specify the requirements for a model transformation. Although, having a dedicated language would be necessary for a rigourous development of a transformation. The definition of this language is outside the scope of this paper. Therefore, we currently express them directly in Alloy.

For the UMLCD_2_RDBMS transformation we list out a set of intents of the tester and their corresponding test model objectives in Table 1. The set of tester's intents are general requirements for the transformation UMLCD_2_RDBMS. However, one may also specify application specific requirements with very specific attributes and classes pertaining to a domain. We do not discuss the specification of application specific constraints, however the expression is very similar to the objectives presented in Table 1.

A thorough discussion about all requirements of UMLCD_2_RDBMS is given in [5]. Each of the test model objectives is extracted to test one of these requirements.

We use Alloy to represent the constraints presented in the objectives. For instance, objective number 6 in Table 1 can be written as follows:

```
fact testers_requirement6
```

No.	Tester's Intent	Test Model Objective
1	Transforming attribute to a single column with the same type	Some Class objects with at least one Attribute of a PrimitiveDataType
2	Transforming a class to top-level	Some Class objects with at least one non-persistent Class attribute
3	Transforming a class to top-level	Some Association objects with destination is a non-persistent Class object
4	Transforming to one/more columns created using persistent classes primary key attribute	Some Class objects with at least one persistent Class attribute
5	Inheritance hierarchies. Only topmost class must be transformed to a table.	Some Class objects with parents
6	Transforming persistent classes	Some Class objects which have is_persistent = true
7	Duplicate keys with same name	Some Class objects with at least one Class attribute with is_primary=True and same name
8	Sub-class attribute must override parent attribute	Some Class objects with parents with attributes having the same name as attributes in the Class object

Table 1. Test Model Objectives and Tester's Intents

```
{some c:Class|c.is_persistent=true}
```

2.4 Partitions on the Input Domain

Category partition testing is widely used in traditional software testing to guarantee input domain coverage of a function based on some partition heuristics. In previous work [10], we have studied how category-partition testing can be applied to model transformation testing. Given an input meta-model (Ecore model only) for a transformation, it is possible to define partitions on domains of all properties of a meta-model (cardinality of references or domain of primitive types for attributes). Then, we have defined several 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 properties that must be satisfied by at least one model in a set of test models.

We have developed a tool called MMCC (Meta-model Coverage Checker) that can generate model fragments, according to a particular criterion, from any meta-model. 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. The automatic generation of new models is not tackled by MMCC.

In this paper, we use the model fragments generated by MMCC for the UMLCD meta-model. These fragments are shown in Figure 4 are used as additional constraints to automatically select test models. For example, MF1 is a model fragment that requires that there exists at least one test model that contains one Classifier with an empty name ($name =$" ") and another Classifier with a non-empty name.

```
MF1{Classifier(name="") and Classifier(name=".+")}
MF2{Class(is_persistent = true) and Class(is_persistent = false)}
MF3{Class(parent = 0) and Class(parent = 1)}
MF4{Class(attrs = 1)and Class(attrs > 1)}
MF5{Attribute(is_primary = true) and Attribute(is_primary = false)}
MF6{Attribute(name="") and Attribute (name=".+")}
MF7{Attribute(type=0) and Attribute (type=1)}
MF8{Association (name="") and Association (name=".+")}
MF9{Association(dest=0) and Association(dest=1)}
MF10{Association(src=0) and Association(src =1)}
MF11{classModel(classifier=1) and classModel(classifier>1)}
MF12{classModel(association =1) and classModel(association >1)}
```

Figure 4. Model Fragments from UMLCD Ecore model

MMCC generated a total of 12 model fragments for the UMLCD Ecore model.

3. Cartier Environment

The Cartier software environment is conceptualized to develop a model transformation testing framework. In this paper we focus on the part where we transform knowledge/constraints in multiple formalisms expressed by different people to one common constraint language for the purpose of selecting test models. First to summarize, we have the following sources of knowledge:

1. Meta-model as an Ecore model with OCL constraints on the Ecore model

2. Model transformation pre-condition as OCL constraints on the Ecore model
3. Partitions of meta-model as sets of objects with properties expressed in Model Fragments language.
4. Test model objectives as sets of objects with properties which are currently expressed as Alloy predicates/facts. Eventually, these will be expressed in a specific test requirements language.

The Cartier environment transforms the above knowledge to a common constraint language which is Alloy [13]. Alloy is based on first-order relational logic and has well defined syntax and semantics. An overview of the transformation framework for Cartier framework is shown in Figure 5. The focus of this paper is the test model selection box highlighted in the figure. In the Figure 5 the large ellipse represents the set of all models for all languages. The smaller ellipse in this large set represents a subset of models that conforms to the constraints that come from multiple sources of testing knowledge. A solution to the relational logic program is obtained by transforming it to a Boolean formula in conjunctive normal form (CNF). The CNF is solved using a satisfiability solver (SAT) solver. The result is a model that is a point in the smaller ellipse. Execution of the transformation on this input model leads to an output model which is again an element of the subset of all possible output models. The output model must conform to its meta-model as shown in the Figure 5. If not, there is an error in the transformation and it must be debugged.

In this paper we present algorithmic pointers to automate the transformation from the different sources of knowledge to Alloy, however at the time of writing most transformations are performed manually. However, ongoing research and tool development on transforming UML models to Alloy is discussed in [2].

In the following subsections we start with an overview of Alloy leading to descriptions of transformations of each source of knowledge to Alloy.

3.1 Alloy Overview

Alloy [13] [1] is a declarative modelling language based on first-order relational logic. The elements in the Alloy world are composed of *Signature*, *Relations*, *Facts*, and *Predicates*. A *Signature* represents the types of objects in a system. Each object of a *Signature* can be related to an other object of the same or different *Signature* via a *Relation*. The *Facts* and *Predicates* state laws/constraints on the *Relation*s and *Signature*s already described.

The Alloy language comes with an analyzer. The semantics of an Alloy model is a *model instance* that satisfies the constraints on the Signatures, Facts, and Predicates. Several model instance (if they exist), can be obtained one after

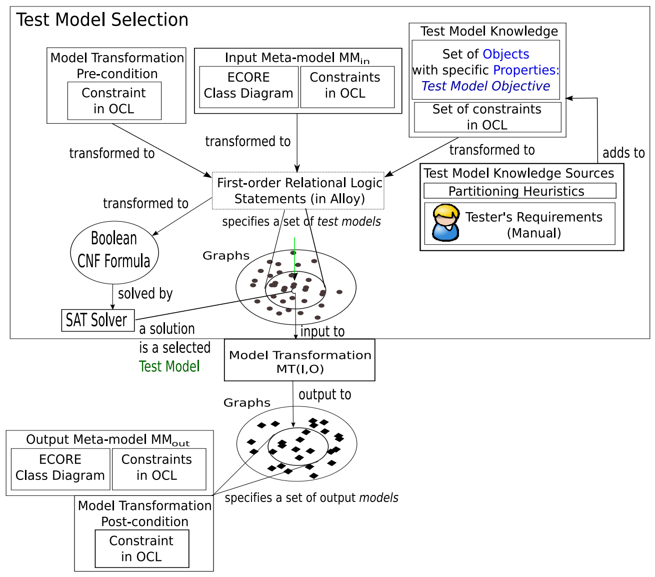

Test Model Selection
Model Transformation Pre-condition
Constraint in OCL
Input Meta-model MM$_{in}$
ECORE Class Diagram
Constraints in OCL
Test Model Knowledge
Set of Objects with specific Properties: Test Model Objective
Set of constraints in OCL
transformed to
transformed to
transformed to
adds to
First-order Relational Logic Statements (in Alloy)
Test Model Knowledge Sources
Partitioning Heuristics
Tester's Requirements (Manual)
transformed to
Boolean CNF Formula
specifies a set of test models
Graphs
solved by
SAT Solver
a solution is a selected Test Model
input to
Model Transformation MT(I,O)
output to
Graphs
Output Meta-model MM$_{out}$
ECORE Class Diagram
Constraints in OCL
Model Transformation Post-condition
Constraint in OCL
specifies a set of output models

Figure 5. Cartier Design Overview

another. A symmetry breaking rule ensures that each model instance is significantly different in the sense of graph isomorphism compared to the previous instance. An Alloy model or a formula is first transformed to a Boolean formula in *conjunctive normal form* (CNF). A SAT solver is then used to solve the Boolean formula. The solution is returned to a graphical representation of a model instance.

The analyzer allows the tester to specify several parameters to introduce more knowledge into model instances. These are expressed as execution parameters or facts. The parameters include something quite general such as the scope or maximum size for each signature/type in the desired model instance to the exact number of specific objects. Constraints in the form of facts can be used to specify a large portion of the target model instance. We use these features of Alloy for selection of test models.

3.2 Transformation of the Meta-model

3.2.1 Basic elements of an Ecore model

First, we discuss the transformation of an Ecore model to Alloy. The Alloy model first starts with a module definition and loading of basic data types. For instance, the UMLCD Alloy model with start with:

```
module tmp/simpleUMLCD
open util/Boolean as Bool
```

The transformation of a class in the Ecore model is performed by transforming a class in Ecore to a *signature* in

Alloy. For instance, the Attribute class in the meta-model shown in Figure 2 is transformed to the following Alloy formula:

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

An Ecore attribute can be transformed to an Alloy *field* of type Bool, Natural or an Int. The is_primary attribute is transformed to a field is_primary of type Bool in Alloy. However, the name attribute which is originally a string in the Ecore model is transformed to a field of type Natural. It is important to note that Alloy does not support string and float fields. The simple reason being the explosion of search space due to the variety of combinations for an ASCII natural/integer representation of characters in a string for example. The focus of Alloy is the model structure and the abstract design of a system. However, to emulate a string field we use a *dictionary* that maps a natural number to a string or a float. The feasible and infeasible conditions of this mapping are:

- A finite number of strings and floats can be defined
- The solver cannot modify the value of string/float for its natural number key
- One cannot specify constraints on the string/float properties
- Equality constraint can be imposed on a natural number key for string/float

A reference to another class is transformed to an Alloy field with one of the specifiers one, lone, or set. which states that there can be exactly one reference, zero or one references or a arbitrary set of references respectively. For instance in the Alloy model, an Attribute has a field type related to the signature Classifier. The one specifier says that there can be exactly one type for an Attribute.

The inheritance relationship between two classes is mapped on to Alloy using *extends*. For instance, Class inherits from Classifier is represented as follows:

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

3.2.2 Containment

The containment relationship in an Ecore model cannot be directly expressed using signatures and fields in Alloy. The containment relationship is transformed to a fact in Alloy. For instance, the containment of Attributes in a Class is given by the following fact:

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

3.2.3 Variable Multiplicity of References

Sometime the specifiers *one*,*lone*, or *many* are not sufficient to describe a relationship. For an arbitrary multiplicity of $[n..m]$ we need to include a fact. For instance, for the signature representation of ClassModel we specify the fact that the number of classifiers in the ClassModel is between 2 and 5.

```
sig ClassModel {
classifier: set Classifier,
association: set Association
}
fact betweenNandMconstraint {
all c:ClassModel | #c.classifier > 2
and #c.classifier < 5}
```

3.2.4 OCL Constraints

OCL constraints are transformed to Alloy facts. There, are several challenges in automating such a process. These challenges are discussed in [2]. For instance the constraint for no cyclic inheritance in Figure 3 results in the following fact:

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

In Section A,the appendix, we present the complete set of Alloy facts transformed from the original OCL constraints.

3.3 Transformation of Model Transformation Pre-condition

The model transformation pre-condition expressed in OCL is transformed to an Alloy fact. The pre-condition that each class in an input model must have at least one primary attribute is necessary for indexing is expressed in the following fact:

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

3.4 Transformation of Test Model Objectives

Test model objectives are all transformed to Alloy facts or Alloy run statements. The set of all the facts is given in the appendix in Section A. However, one can also specify a run command such as:

```
pred example() {}
run example for  20
```

The above run command is a generic run command which states that the graph depth of the model instance solution is 20. However, one can even specify the exact number of objects using the *exactly* prefix after the predicate declaration (*example* in this case).

3.5 Transformation of Partitions

Partition knowledge in form of model fragments discussed in Section 2.4, is transformed to Alloy facts. For instance, the partition:

Classifier(*name* =“ ”) and Classifier(*name* =“.+”)

is transformed to the fact:

```
fact partition1
{some c1:Classifier, c2:Classifier|
c1.name=0 and c2.name!=0
/*0 is null, non-zero others*/}
```

4 Experiments

We select test models from the input domain of the UMLCD_2_RDBMS transformation using the different sources of testing knowledge we have already discussed. We show the selection of four UML Class Diagram (UMLCD) models.

To begin, we use the Alloy analyzer to generate a model that conforms only to the UMLCD meta-model. This is shown in Figure 6 (a) using UMLCD concrete syntax. The selected test model was found in a *scope* of 10. The scope is the maximum number of objects for each type (or class) in the meta-model. The model selection is performed up to the limit proposed by the scope. We see that the resulting model satisfies all meta-model constraints. However, an attribute of Class0 is not primary. This implies that it is not a valid input to UMLCD_2_RDBMS.

The UMLCD_2_RDBMS must get as input an UMLCD that has Class objects with at least one primary attribute. This is essential for generating valid index-able RDBMS models. To take this issue into account we introduce the model transformation pre-condition for UMLCD_2_RDBMS. The resulting model is shown in Figure 6 (b). The selected model has classes with at least one primary attribute just as required by the pre-condition for UMLCD_2_RDBMS. The selected model was found in a maximum scope of 20. We note that the model now has two classes Class6 and Class7, both of which have at least one primary attribute. This new aspect in the test model now will allow the tester to generate UMLCD models that can be serialized to RDBMS.

Sources of Test Knowledge	Time(sec)
Meta-model	0.78
Meta-model + Pre-condition	7.813
Meta-model + Pre + TMO	7.97
Meta-model + Pre + MFs	10.477

Table 2. Test Model Selection Times

Next, we perform an experiment with test model objectives. In addition to the meta-model and the model transformation pre-condition we introduce a test model objective. We particularly want to select a test model that has some classes with $is\_persistent = True$. This is objective 6 in the Table 1. We select a model in a maximum scope of 20. The resulting model is shown in Figure 6 (c). We note the class Class5 is persistent as per the tester's objective. The existence of persistent classes in the test model will now allow a tester to test the persistence implementation of the UMLCD_2_RDBMS.

Finally, we introduce model fragment facts along with the meta-model and pre-condition. The model that covers the meta-model and five model fragments is shown in Figure 6 (d). The resulting model covers some of the model fragments facts we generated from the Ecore model. The model is selected for a maximum scope of 20. The model fragments covered ,as described in Figure 4, were MF2, MF3, MF4, MF5. This guarantees that the equivalence classes for property values are covered at least once by a test model. In terms of test qualification, this increases the trust we have in the test models, based on input domain coverage.

In Table 2, we summarize the time taken (on a P4 2.6Ghz desktop, with 1Gb RAM) to select test models. From the table we can generally say that the more testing knowledge we have the longer it takes to obtain test models. However, for the price paid, the quality of test model is much higher with more information to detect bugs in model transformations. For instance, adding a test model objective such as at least one class with $is\_persistent = True$ makes the test model execute the persistence implementation in a model transformation.

5 Related Work

Techniques for model transformation validation have been proposed using formal verification and testing. We focus on related work in the domain of testing model transformations. In Fleurey et al. [11], the authors describe the problem of testing model transformations and explain test adequacy criteria for test models. There are two standard

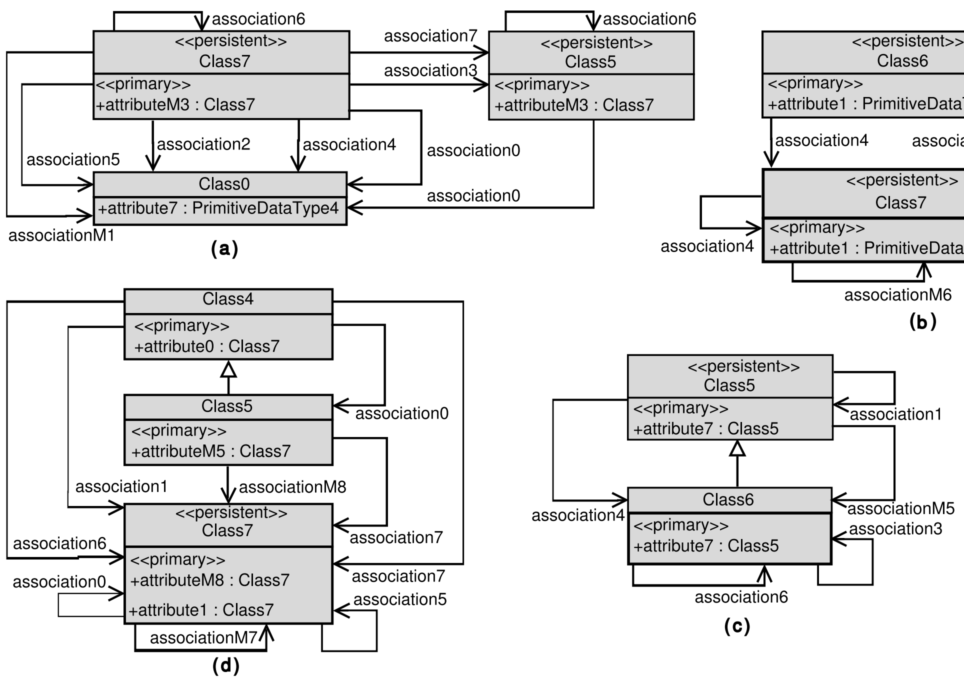


Figure 6. (a) Model conforming to Meta-model (b)Model conforming to Meta-model + Pre-condition (c) Model conforming to Meta-model+ Pre-condition + Test Model Objective (d) Model conforming to Meta-model + Pre-condition + Model Fragment

approaches for testing model transformations : white-box and black-box testing.

White-box testing has been studied for model transformation testing in Kuster et al. [15]. However, the heterogeneity of model transformation languages makes it increasingly hard to develop white box testing methodologies. This is primarily because a testing tool needs to be constructed for every new language. This is extremely expensive for transformations among domain specific languages. This is one of the main reasons we choose black-box testing as a means to large-scale testing of model transformations.

In the black-box testing of programs (including model transformations) a set of test models are synthesized to cover input domain and test model objectives. Model synthesis is the first requirement for black-box testing. In Ehrig et al. [9], the authors propose a graph grammar based approach to generate models that conform to a class diagram (or Ecore model). These models do not conform to any OCL constraints on the meta-model. Similarly, Brottier el al. [7] present an imperative algorithm to synthesize models that conform only to the Ecore model. The key issue is to generate test models that not only conform to an Ecore model or a class diagram but several other complex constraints emerging from different sources such as OCL, input domain partitions, and test model objectives.

In software testing the Korat (Chandra et al.) [6] system for automatic testing of Java programs prunes large input search spaces using knowledge from pre-condition predicates in Java. Korat can deal with input test cases that are bounded such as those implemented in the Java Collections Framework. The Korat framework however does not present formal semantics for its predicates. It also cannot synthesize a constrained data structure that represents a model conforming to a modelling language. Going beyond standard data structures to models expressed using an arbitrary modelling language is the focus of our work. Moreover, selection of models for testing model transformations poses a large-scale constraint satisfaction problem that cannot be solved by existing nature inspired techniques such as bacteriologic algorithms and genetic algorithms [4].

In Sen et al.[19] we propose a Prolog based methodology to combine constraints from different sources to select test models. However, Prolog is based on first-order horn clause logic and hence it is quantifier free (a constraint of a set of objects cannot be specified). In response to this issue we were inspired by ongoing research in transforming UML models to Alloy as described by Kyriakos et al. [2]. Alloy is a first-order relational logic language with support quantifiers and transitive closure. These features allow the specification of constraints on a set of objects rather than a specific object. As a consequence of this transformation the same group has presented results on static analysis of model transformation specifications (Kyriakos et al.) [3]. We outline a similar transformation from Ecore to Alloy but our main consideration and application (of Alloy) is the combination of knowledge from several different formalism for the purpose of selecting test models for model transformation testing.

Among other approaches to model transformation testing we notably encounter mutation analysis based approaches. Graph transformations have been widely used to design and develop model transformations. For instance, in Darabos et al. [8], the authors present a mutation analysis based approach to test graph transformations. Graph pattern matching is used to inject faults into graph transformation rules and a predefined test set is used to find these faults. Another mutation analysis based technique is given in Mottu et al. [16]. The authors discuss a set of mutation operators for textual transformation languages. However, in both cases the problem of defining a set of test models still remains.

An important advantage with regard to software reliability of test model selection could be the development of an analog of *reliable objects* [14] in MDE. The authors present a method to embed test cases into components to make them self-testable.

6. Conclusion

This paper proposes a novel approach for test data selection in the context of model transformation. This approach for selection leverages different sources of knowledge that can be produced during model transformation development and specifically for test data generation. We combine those different sources of knowledge into a common model that specifies a set of constraints that should be satisfied by the test models.

We have outlined a tool, called Cartier, that uses Alloy for constraint resolution. Cartier allows combining one or several sources of knowledge. The experiments show that the knowledge from different sources such as pre-conditions, test model objectives and partitions can all be combined to synthesize test models.

As part of our future planning we intend to perform *mutation analysis* with the synthesized test models. We plan to use the mutation operators defined in Mottu et al. [16]. In particular, we would like to experiment which source of knowledge or which combinations produce the most useful test models in terms of error detection capabilities.

7 Acknowledgment

This work is a part of the DOMINO project, granted by the French National Research Agency (ANR).

References

[1] http://alloy.mit.edu.

[2] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. Uml2alloy: A challenging model transformation. In *MoDELS*, pages 436–450, 2007.

[3] K. Anastasakis, B. Bordbarand, and J. M. Kuster. Analysis of model transformations via alloy. In *MoDeVVa'07 associated with MoDeLs*, Nashville, Tennessee, October 2007.

[4] B. Baudry, F. Fleurey, J.-M. Jezequel, and Y. L. Traon. Automatic test cases optimization using a bacteriological adaptation model: Application to . net components. In *International Conference on Autoamated Software Engineering*, Edinburgh, September 2002.

[5] J. Bezivin, B. Rumpe, A. Schurr, and L. Tratt. Model transformations in practice workshop, october 3rd 2005, part of models 2005. In *Proceedings of MoDELS*, 2005.

[6] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, 2002.

[7] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *Proceedings of ISSRE'06*, Raleigh, NC, USA, 2006.

[8] A. Darabos, A. Pataricza, and D. Varro. Towards testing the implementation of graph transformations. In *GT-VMT workshop associated to ETAPS'06*, pages 69 – 80, Vienna, Austria, April 2006.

[9] K. Ehrig, J. Küster, G. Taentzer, and J. Winkelmann. Generating instance models from meta models. In *FMOODS'06 (Formal Methods for Open Object-Based Distributed Systems)*, pages 156 – 170., Bologna, Italy, June 2006.

[10] F. Fleurey, B. Baudry, P.-A. Muller, and Y. L. Traon. Towards dependable model transformations: Qualifying input test data. *Software and Systems Modelling (Accepted)*, 2007.

[11] F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: Testing model transformations. In *MoDeVa'04 (Model Design and Validation Workshop associated to ISSRE'04)*, Rennes, France, November 2004.

[12] B. Frank. *Eclipse Modeling Framework*, volume 1 of *The Eclipse Series*. Addison-Wesley, 2004.

[13] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, March 2006.

[14] J.-M. Jezequel, D. Deveaux, and Y. L. Traon. Reliable objects: a lightweight approach applied to java. *IEEE Software*, 18(4):76–83, 2001.

[15] J. Küster and M. Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In *MoDeVa'06 (Model Design and Validation Workshop associated to MoDELS'06)*, Genova, Italy, October 2006.

[16] J.-M. Mottu, B. Baudry, and Y. L. Traon. Mutation analysis testing for model transformations. In *Proceedings of ECMDA'06 (European Conference on Model Driven Architecture)*, Bilbao, Spain, July 2006.

[17] OMG. MOF 2.0 Core Final Adopted Specification. In *http://www.omg.org/cgi-bin/doc?ptc/03-10-04*, 2005.

[18] OMG. The Object Constraint Language Specification 2.0, OMG Document: ad/03- 01-07, 2007.

[19] S. Sen, B. Baudry, and D. Precup. Partial model completion in model driven engineering using constraint logic programming. In *International Conference on the Applications of Declarative Programming*, 2007.

A. Combined Alloy Model for UMLCD

```
module tmp/simpleUMLCD
open util/Boolean as Bool
//Meta-model Entities
sig ClassModel{classifier: set Classifier,
association: set Association}
abstract sig Classifier {name : one Natural}
sig PrimitiveDataType extends Classifier {}
sig Class extends Classifier
{is_persistent: one Bool,parent : lone Class,
attrs : some Attribute}
sig Association {name: Natural,
dest: one Class,src: one Class}
sig Attribute{name:Natural,is_primary:Bool,
type: one Classifier}
//Meta-model constraints//
fact noCyclicInheritance {no c: Class |
c in c.^parent}
fact uniqueAttribNames{all c:Class|all a1:c.attrs,
a2 : c.attrs |a1.name==a2.name => a1 = a2}
fact attributeContainment{all c1:Class, c2:Class|
all a1:c1.attrs, a2:c2.attrs | a1==a2 => c1=c2}
fact oneClassModel {one ClassModel}
fact classifierContainment {all c:Classifier|
c in ClassModel.classifier}
fact associationContainment {all a:Association|
a in ClassModel.association}
fact uniqueClassifierName {all c1:Classifier,
c2:Classifier | c1.name==c2.name => c1=c2}
fact uniqeNameAssocSrc {all a1:Association,
a2:Association | a1.name == a2.name =>
(a1 = a2 or a1.src != a2.src)}
//Test Model Objectives
fact testers_requirement1 {some a:Attribute|
a.type = PrimitiveDataType}
fact testers_requirement2{some a:Attribute|
a.type.is_persistent=False}
fact testers_requirement3{some a:Association|
a.dest.is_persistent = False}
fact testers_requirement4{some a:Attribute|
a.type.is_persistent=true}
fact testers_requirement5{some Class.parent}
fact testers_requirement6
{some c:Class|c.is_persistent=true}
fact testers_requirement7
{some c:Class |some a:c.attrs|a.name=c.name
and a.type=Class and a.is_primary=true}
fact testers_requirement8
{some cA:Class | some aA :cA.attrs,
pA:cA.parent.attrs|aA.name==pA.name}
/*UMLCD to RDBMS Pre-condition*/
fact atleastOnePrimaryAttribute {
all c:Class|one a:c.attrs|a.is_primary==true}
//Partition Requirements
fact partition1 {some c1:Classifier,
c2:Classifier| c1.name=0 and c2.name!=0}
fact partition3
{some c1:Class,c2:Class|c1.is_persistent=true
 and c2.is_persistent = False}
fact partition4
{some c1:Class,c2:Class|
#c1.parent=0 and #c2.parent=1}
fact partition5{some c1:Class,c2:Class,c3:Class|
#c1.attrs=0 and #c2.attrs=1 and #c3.attrs>1}
fact partition6{some a1:Attribute,
a2:Attribute|a1.is_primary=true and
a2.is_primary=False}
fact partition7{some a1:Attribute,a2:Attribute|
a1.name=0 and a2.name!=0}
fact partition8{some a:Attribute|
#a.type=1 and a.name=1}
fact partition9{some a1:Association,
a2:Association|a1.name=0 and a2.name!=0}
fact partition10
{some a1:Association,a2:Association|
#a1.dest=0 and #a2.dest=1}
fact partition11{some a1:Association,
a2:Association|#a1.src=0 and #a2.src=1}
fact partition12
{some c1:ClassModel,c2:ClassModel,
c3:ClassModel|#c1.classifier=0 and
#c2.classifier=1 and #c3.classifier>1}
fact partition13{some c1:ClassModel,
c2:ClassModel,c3:ClassModel|
#c1.association=0 and #c2.association=1
and #c3.association>1} pred example() {}
run example for 20
```