Using Models of Partial Knowledge to Test Model Transformations

Sagar Sen[1], Jean-Marie Mottu[2], Massimo Tisi[1], and Jordi Cabot[1]

[1] AtlanMod, École des Mines de Nantes - INRIA, France
{sagar.sen, massimo.tisi, jordi.cabot}@inria.fr
[2] Université de Nantes - LINA (UMR CNRS 6241)
jean-marie.mottu@univ-nantes.fr

Abstract. Testers often use partial knowledge to build test models. This knowledge comes from sources such as requirements, known faults, existing inputs, and execution traces. In Model-Driven Engineering, test inputs are models executed by model transformations. Modelers build them using partial knowledge while meticulously satisfying several well-formedness rules imposed by the modelling language. This manual process is tedious and language constraints can force users to create complex models even for representing simple knowledge. In this paper, we want to simplify the development of test models by presenting an integrated methodology and semi-automated tool that allow users to build only small partial test models directly representing their testing intent. We argue that partial models are more readable and maintainable and can be automatically completed to full input models while considering language constraints. We validate this approach by evaluating the size and fault-detecting effectiveness of partial models compared to traditionally-built test models. We show that they can detect the same bugs/faults with a greatly reduced development effort.

1 Introduction

Model transformations are core components that automate important steps in Model-Driven Engineering (MDE), such as refinement of input models, model simulation, refactoring for model improvement, aspect weaving into models, exogenous/endogenous transformations of models, and the classical generation of code from models. Models and transformations have a widespread development in academia and industry because they are *generic* artifacts to represent complex data structures, constraints, and code abstractions. However, there is little progress in techniques to test transformations [5]. Testing requires the specification of software artifacts called *test models* that aim to detect faults in model transformations. Specifying test models manually is a tedious task, complicated by the fact that they must conform to a modelling language's specification and numerous well-formedness rules. For instance, the specification of the UML contains numerous inter-related concepts and well-formedness rules for its models such as class diagrams, activity diagrams, and state machines. The issue becomes crucial when a tester needs to create several hundred test models for a model transformation.

The knowledge to create test models can come from various sources. Usually some tests have a direct correspondence with application requirements, others are conceived

by imagining corner cases that can cause an error in a transformation. Several methods exist to derive further testing knowledge. For instance, analyzing a model transformation can reveal locally used classes, properties and possibly some of their values or bounds on values called a *footprint* [16]. Similarly, analyzing a localized fault via techniques such as dynamic tainting [12] in a model transformation can reveal patterns in the input modelling language that evoked the fault. Other sources include existing models via model slicing [7] and execution traces of a model transformation [3]. However, most of this knowledge is incomplete or *partial* in the sense that it must be completed and certified as a valid input test model that conforms to the well-formedness rules and constraints of the modelling language. We face and rise to three challenges:

Challenge 1: How can we express partial knowledge in a modelling language? We call the artifact containing this knowledge a *partial model*.

Challenge 2: How can we automatically complete a partial model?

Challenge 3: Are these automatically completed models effective in detecting the same faults that a human-made model containing the partial knowledge detects?

In this paper, we provide a methodology to generate effective test models from partial models and a semi-automated supporting tool [25]. The methodology to generate complete test models from partial knowledge is divided into two phases. In the first phase, we need to specify partial model(s) as required by *Challenge 1*. We propose to represent a partial model as a model conforming to a relaxed version of the original input metamodel of the model transformation. The relaxed metamodel allows specification of elements in a modelling language without obligatory references, containments, or general constraints that were in the original metamodel. Our tool adopts the transformation in [24] to generate a relaxed metamodel suitable to specify a partial model. In the second phase, we automatically complete partial models by integrating our tool PRAMANA [26] as required by *Challenge 2*. PRAMANA transforms the input metamodel to a base constraint satisfaction problem A_b in Alloy [15]. In this paper, we re-write partial models as *predicates in Alloy* that are juxtaposed to A_b. We also specify finite bounds on the satisfaction problem called the *scope*. The scoping strategy can be modified depending on whether we would like to generate minimally sized or large test models. We solve the constraint satisfaction problem in Alloy using a SAT solver to generate one or more test models that complete the partial models and satisfy all well-formedness rules of the input modelling language. If the partial model conflicts with a modelling language constraint, which has higher priority, we do not generate a test model, but we give feedback to the tester about the partial model being invalid. Our approach is applicable to transformations written using model transformation languages (e.g., ATL [17], Kermeta [22]) or also general-purpose languages (e.g., Java).

Finally, we experimentally evaluate a case study, to tackle the last *Challenge 3*, but also to show the benefits of our methodology in drastically reducing the manual aspects of test specification in MDE. Our experimentation shows that a set of small partial models can detect all the faults that human-made complex models detect. We compare, by using *mutation analysis* for model transformations [11] [21], the bug-detecting effectiveness between a test set of completed partial models and human-made test models. In our experiments, we employ the representative case study of transforming simplified UML class diagram models to database (RDBMS) models called class2rdbms. Mutation

analysis on this transformation reveals that a set of 14 concise partial models can detect the same bugs as 8 complex man-made models. The 8 complete models contain more than twice (231 vs. 109) the number of elements compared to partial models. It is also important to note that the complete models were constructed meticulously to satisfy well-formedness rules while the partial models contain loosely connected objects that are automatically linked by our tool. The partial models are noise-free in the sense that they illustrate the *testing intent* precisely without the clamor of well-formedness rules. This result suggests that concise expression or extraction of partial knowledge for testing is *sufficient*, *comparatively effective*, and *less tedious* than manually creating test models hence solving *Challenge 3* and promoting our approach.

The paper is organized as follows. In Section 2 we present the representative case study for transformation testing. In Section 3 we present our integrated methodology to generate complete test models from partial knowledge. In Section 4 we present our experimental setup and results. In Section 5 we discuss related work. We conclude in Section 6.

2 Case Study

In the paper, we consider the case study of transforming simplified UML Class Diagram models to RDBMS models called class2rdbms. We briefly describe class2rdbms in this section and discuss why it is a representative transformation to validate our approach.

For testing a model transformation, the user provides input models that conform to the input metamodel *MM* (and possibly transformation pre-condition $pre(MT)$). In Figure 1, we present the simplified UMLCD input metamodel for class2rdbms. The concepts and relationships in the input metamodel are stored as an Ecore model [10] (Figure

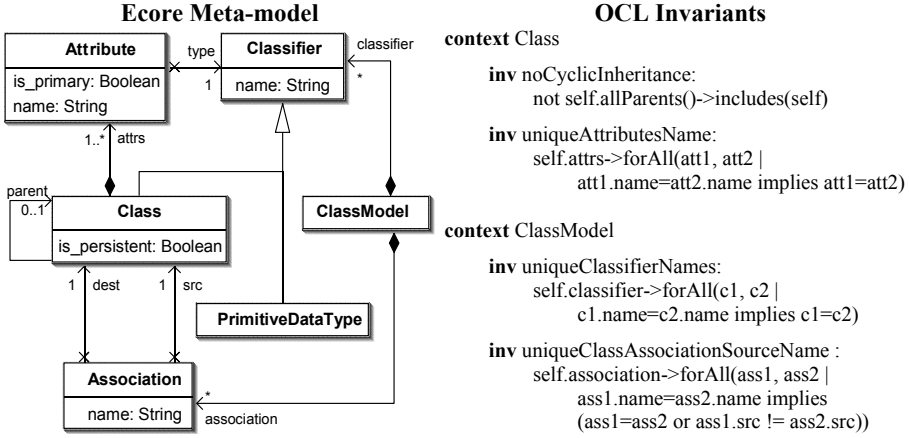

Fig. 1. (a) Simplified UML Class Diagram Ecore Metamodel (b) OCL constraints on the MM

1 (a)). Part of all the invariants on the simplified UMLCD Ecore model, expressed in Object Constraint Language (OCL) [23], are shown in Figure 1 (b). The Ecore model and the invariants together represent the true input metamodel for class2rdbms. The OCL and Ecore are industry standards used to develop metamodels and specify different invariants on them.

For this paper we use the Kermeta implementation of class2rdbms, provided in [21]. The transformation class2rdbms serves as a sufficient case study for several reasons. The transformation is the benchmark proposed in the MTIP workshop at the MoDELS 2005 conference [6] to experiment and validate model transformation language features. The input domain metamodel of simplified UMLCD covers all major metamodelling concepts such as inheritance, composition, finite and infinite multiplicities. The constraints on the simplified UMLCD metamodel contain both first-order and higher-order constraints. There also exists a constraint to test transitive closure properties on the input model such as there must be no cyclic inheritance. The transformation exercises most major model transformation operators such as navigation, creation, and filtering (described in more detail in [21]) enabling us to test essential model transformation features.

3 Methodology

We present a methodology to generate complete models from partial models, and the supporting tool [25] we developed. We describe the process in three phases: 1) Partial Model Specification, 2) Transformation to ALLOY, 3) Model Completion.

1) Partial Model Specification

A partial model is essentially a graph of elements such that: (1) The elements are instances of classes in the modelling language metamodel *MM* (2) The partial model does not need to conform to the language metamodel *MM* or its invariants expressed in a textual constraint language such as OCL. A complete model on the other hand contains all the objects of the partial model and additional objects or property value assignments in new/existing objects such that it conforms both to the metamodel and its invariants.

In the first phase, the user can specify partial models using an automatically-generated relaxed language. Given an input metamodel *MM* we generate a relaxed metamodel MM_r using a relaxation transformation as shown in Figure 2 (a). The transformation is adopted from Ramos et al. [24]. The relaxed metamodel MM_r allows the specification of partial models that need not satisfy a number of constraints enforced by the original metamodel *MM*. In Figure 2 (b), we show the relaxed metamodel derived from the simple class diagram metamodel shown in Figure 1 (a). The relaxed metamodel allows the specification of partial models in a modelling language. For instance, in Figure 4(a), we show a partial model specified using MM_r. It is important to clarify that PObject acts as a linking object to another metamodel that allows writing of patterns called *model snippets* on the relaxed metamodel. for a pattern-matching framework.This pattern matching metamodel from Ramos et al.[24] is not used in this paper and is hence not shown. It is interesting to note that the partial model specified using MM_r allows specification

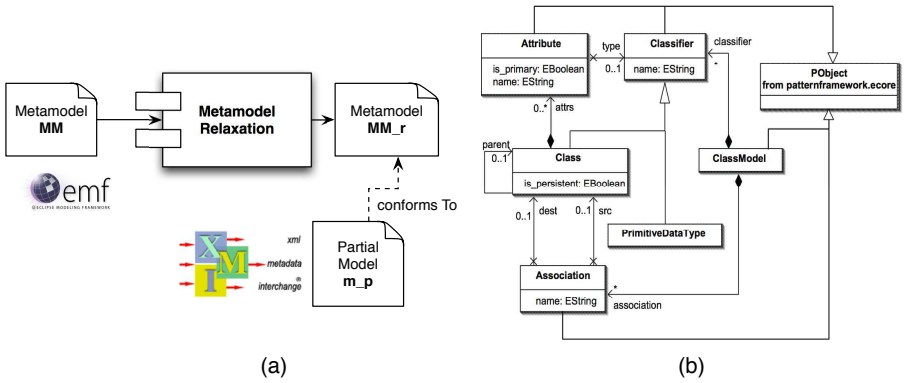

Fig. 2. (a) Metamodel relaxation to help specifying a partial model (b) Relaxed metamodel MM_r

of objects required for testing. For instance, we specify Class and Association without the need to specify their containment in ClassModel. Similarly, a full specification of all property values is not required. The only properties specified in the partial model are Class.is_persistent, Association.src, Association.dest, and Class.parent. On the contrary, a model that conforms to *MM*, must satisfy the metamodel *MM*'s constraints and its well-formedness rules such as no cyclic inheritance such as in Figure 1(a), (b). Using OCL [23] constraints is another way to specify partial models. However, it is more complex for the tester to write an OCL constraint equivalent to a partial model: it is based on the complete *MM* to navigate the model from its root to the constrained concepts. For instance, whereas a partial model constrains one class to have its attribute `is_attribute = true` is made only with this class with only this attribute initialized, the same constraint written in OCL is:

```
input.classifier.select(c|c.oclIsType(Class)).exists(cs|cs.attrs.exists(a|a.is_primary = true))
```

In our approach, the concepts in the partial model are those that are available in *MM* but no one is mandatory.

2) Transformation to ALLOY

In the second phase, we transform the original metamodel *MM* to Alloy [15]. We integrate the tool PRAMANA presented in Sen et. al. [26], for the transformation in [25]. PRAMANA first transforms a metamodel expressed in the Ecore format using the transformation rules presented in [26] to ALLOY. Basically, classes in the input metamodel are transformed to ALLOY signatures and implicit constraints such as inheritance, opposite properties, and multiplicity constraints are transformed to ALLOY facts. Second, PRAMANA does not fully address the issue of transforming invariants and preconditions expressed on metamodels in the industry standard OCL to ALLOY. The automatic transformation of OCL to ALLOY presents a number of challenges that are discussed in [1]. 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

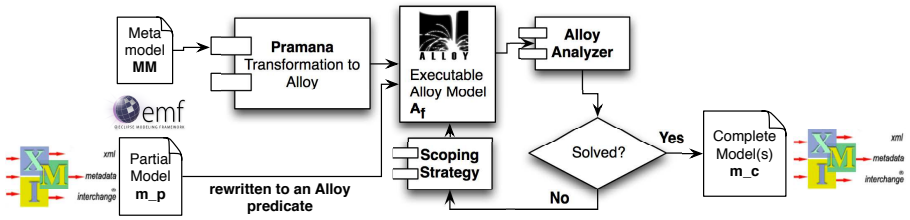

Fig. 3. Generation of Complete Model(s) from a Partial Model

case. The reason being that OCL and ALLOY were designed with different goals. OCL is used mainly to query a model and check if certain invariants are satisfied. ALLOY facts and predicates on the other hand enforce constraints on a model. 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 overly expressive for the purpose of finite-domain constraint solving. In our case study, we have been successful in *manually transforming* all constraints on the simplified UMLCD metamodel to ALLOY from their original OCL specifications. Can we fully automate this step? This question remains to be answered in our approach. However, the manual transformation is a one time mental effort for a fixed set of constraints.

The ALLOY model generated in the previous phase has to be extended with the information coming from the partial test models. A partial model, such as in Figure 4 (a), is *manually re-written* to an ALLOY predicate as shown in Figure 4 (b). For the translation we navigate all objects of a certain type and put them together as an ALLOY predicate. The predicate states that there exists three objects c1,c2,c3 of type Class such that they are unequal, and only c1.is_persistent =True. The Class objects c2 and c3 inherit from c1. There exists also an Association object a1 such that its source a1.src is Class object c2 and destination a1.dest is c3. The name properties of the Class objects c1,c2,c3 and Association object a1 are not specified. They also do not contain a primary attribute which is mandatory for the transformation class2rdbms. The partial model objects also do not have to be contained in a ClassModel object. This process can be automated to generate a concise and effective ALLOY predicate. For instance, in [28], the authors have automated the transformation of partial models specified in a model editor. It can also be improved to consider negative application conditions (false-positives) or bounds on maximum/minimum of objects that can be specified in a partial model.

3) Model Completion

The final phase in our methodology is that of solving the translated ALLOY predicate in the executable ALLOY model A_f to obtain one or more complete models.

Model completion requires finite values such as the upper bound on the number of objects of the classes in the *MM*, or the exact number of objects for each class, or a

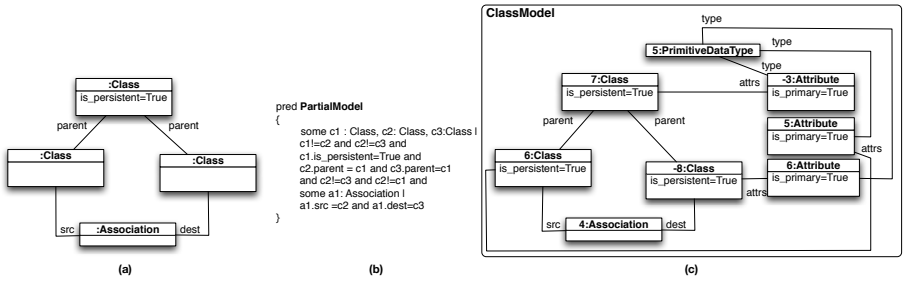

Fig. 4. (a) Partial Model P3 (b) Re-written Alloy Predicate (c) Complete Model from P3

mixture of upper bounds and exact number of objects for different classes. These values are called the *scope*. We call the approach to specify the scope the *scoping strategy*. The default scoping strategy is the upper bound defined by the number of objects of each type in the partial model. For objects that are not part of the partial model we manually fine tune the scope to a minimum value such that a complete model can be generated. The scope of the top-level container class (ClassModel in our case study) is often exactly one. The scoping strategy is generated in the form of a ALLOY *run command* that is finally inserted in the executable ALLOY model A_f as shown in Figure 3. For example, the run command generated and fine-tuned for the partial model in Figure 4 (a) is:

```
run partialModel for exactly 1 ClassModel, 4 int, 3 Class, 3 Attribute, 1 PrimitiveDataType, 1 Association
```

The above run command is *manually fine-tuned* for Attribute and PrimitiveDataType. Fine-tuning is necessary in our case study since all Class objects require at least one primary Attribute object.

The ALLOY model is transformed to a set of expressions in relational calculus by the ALLOY analyzer. These expressions are then transformed to Conjunctive Normal Form (CNF) [29] using KodKod. Finally, the CNF is solved using a SAT solver [19]. The low-level SAT solutions are transformed back to XMI models that conform the initial metamodel. The resulting XMI models are validated by loading them into an EMF model editor. The editor ensures both the satisfaction with respect to an Ecore metamodel and OCL constraint checker. The use of an industrial tool helps ensure that the input models contain objects of valid classes, and conform to all metamodel and OCL constraints

There is always the option to either automatically/manually fine tune the scoping strategy to generate big or concise complete models. For instance, in Figure 4 (c), we show a concise complete model that we generate for the partial model in Figure 4 (a). One or more non-isomorphic solutions can be generated by ALLOY by adding a symmetry breaking constraint for each new solution. The symmetry breaking constraints are generated and added automatically by Alloy whenever a new solution is requested. This in-turn allows us to generate several hundred different models that are non-isomorphic if possible. **All elements of the partial model are preserved in the complete test models. Minimal number of objects are added so that all well-formedness rules**

Table 1. Degree of automation in our tool

Aspect of Tool	Degree of Automation
Metamodel to Alloy in PRAMANA	automatic
OCL to Alloy	manual
Metamodel to relaxed metamodel	automatic
Partial Model Specification	manual
Partial Model to Alloy Predicate	currently manual/can be automated [28]
Solution Scoping	default and manually tunable
Solving Alloy Model from API in PRAMANA	automatic
XMI model instance from solution in PRAMANA	automatic

and other constraints are satisfied. In Table 1, we present the degree of automation achieved in our tool.

4 Experiments

In this section we perform experiments to address two questions:

Q1 Can tests derived from partial models detect the same faults that human-made models detect?
Q2 Are partial models more concise than equivalently powerful human-made models?

The inputs to our experiments are (a) the simplified UMLCD input metamodel from class2rdbms, (b) a set of well-formedness rules and class2rdbms pre-conditions in OCL. The experimentation proceeds as follows:

1. A set of random faults are injected in the class2rdbms model transformation, using a mutation tool;
2. A first modeler, aware of the previous mutations, develops a set of models that kills all the injected faults;
3. A second modeler, aware of the mutations, develops a set of partial models to kill all the injected faults, following our approach;
4. We measure the size of test models and compare the effectiveness of the two test sets.

The two modelers are both experienced in testing and modeling. For our final evaluation, we use a Macbook Pro Intel dual-core processor 2.7 GHz, 8 GB RAM to generate complete models and perform the analysis.

4.1 Injecting Faults in the Model Transformation

Our experimental evaluation is based on the principles of mutation analysis [11]. Mutation analysis involves creating a set of faulty versions or *mutants* of a program. A test set must distinguish the correct program output from all the output of its mutants. In

Table 2. Partition of the class2rdbms mutants depending on the mutation operator applied

Mut. Oper.	CFCA	CFCD	CFCP	CACD	CACA	RSMA	RSMD	ROCC	RSCC	Total
# of Mutants	19	18	38	11	9	72	12	12	9	200

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 injecting a single fault. 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.

To inject faults in our transformation, we use the mutation operators for model transformations presented by Mottu et al. [21]. 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, Mottu et al. defined the following mutation operators:

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.

We apply all these operators on the class2rdbms model transformation. We identify in the transformation code all the possible matches of the patterns described by each operator. For each match we generate a new mutant. This way we produce two hundred mutants from the class2rdbms model transformation with the partition indicated in Table 2.

In general not all injected mutants become faults, since the semantics of some of them is equivalent to the correct program, and therefore they can never be detected. The

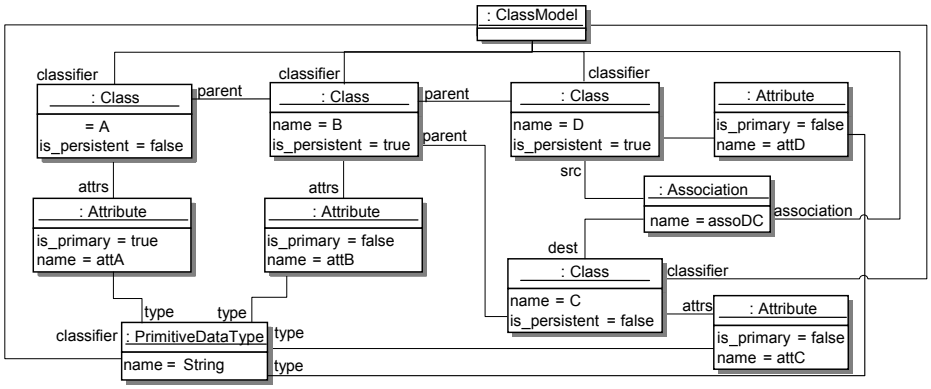

Fig. 5. Human-made Test Model M1

controlled experiments presented in this paper use mutants presented in our previous work [21] where we have clearly identified mutants and equivalent mutants.

4.2 Building Manual Tests

A modeler manually designed a set of test models, by analyzing the 200 faults we injected in class2rdbms as described earlier. The objective was to obtain a test set able to kill all the mutants we generated. The resulting test set is composed by 8 models conforming to the fully constrained UMLCD modelling domain (metamodel + well-formedness rules + pre-conditions for class2rdbms). Together, the hand-made test models count 231 elements. We used mutation analysis to verify that these 8 human-made models can kill all 194 mutants (excluding 6 equivalent mutants identified).

4.3 Building Partial Models

For the experimentation, we could have built the partial models from scratch (as it normally happens) but we prefer to extract partial models from the manual models derived in the previous phase. We need it to have a direct correspondence between traditional models and partial models, that is useful in the evaluation. This does not influence the global experimentation results: from scratch or from a traditional model, the second modeler wants to kill one mutant. With both methods, only the mutant is targeted. Additionally with our experimental approach, partial models contain elements comparable with traditional models.

We incrementally derive our partial models from the 8 human-made models. Given a reference human-made model (as shown in Figure 5) we extract a partial model (in Figure 4 (a)). A partial model is extracted from a human-made model by keeping only the useful elements to kill one target mutant. We use our methodology in Section 3 to automatically complete the partial model to get 10 test models. Second, we apply mutation analysis to obtain a score for the 10 complete test models if the partial model is solvable. If the mutation score is not 100% we extract more partial models from the

Table 3. Size Comparison Between Partial Models and Human-made Models

partial model	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14								
from reference model	M1	M1	M1	M3	M5	M6	M3	M4	M7	M3	M4	M2	M1	M8	M1	M2	M3	M4	M5	M6	M7	M8
#ClassModel	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
#Class	1	3	3	2	2	2	2	2	3	3	4	1	1	3	4	3	3	3	2	3	2	3
#name attr.	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	3	3	3	2	3	2	3
#is_persistent attr.	1	3	1	2	0	0	2	0	0	3	1	0	1	1	4	3	3	3	2	3	2	3
#parent relation	0	1	2	0	1	1	1	0	0	1	1	0	0	1	3	1	2	1	1	1	0	1
#Association	0	0	1	1	0	2	1	2	0	0	2	0	0	1	1	1	1	2	0	3	0	1
#name attr	0	0	1	1	0	2	0	0	0	0	0	0	0	0	1	1	1	2	0	3	0	1
#src relation	0	0	1	1	0	2	1	2	0	0	2	0	0	1	1	1	1	2	0	3	0	1
#dest relation	0	0	0	0	0	0	1	2	0	0	2	0	0	1	1	1	1	2	0	3	0	1
#Attribute	1	0	0	0	2	0	0	0	2	1	0	1	1	0	4	3	3	3	2	3	3	2
#is_primary attr.	1	0	0	0	0	0	0	0	2	1	0	1	1	0	4	3	3	3	2	3	3	2
#name attr.	0	0	0	0	2	0	0	0	0	0	0	0	0	0	4	3	3	3	2	3	3	2
#type relation	0	0	0	0	0	0	0	0	2	1	0	1	1	0	4	3	3	3	2	3	3	2
#PrimitiveDataType	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	1	1	1	1
#name attr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Total #concept	4	7	9	7	7	9	8	8	9	10	12	5	6	8	38	29	30	33	18	37	21	25
Total #concept per set	of partial models:					109									of models:				231			

set of complete models to kill live mutants. We repeat this process until test models generated from partial models give a 100% mutation score.

For instance, to kill a mutant we extract a partial model P3 in Figure 4 (a) from the human-made model M1 in Figure 5. The human-made model contains many concepts (classes, classes' attributes and relationships), it satisfies the invariants of the meta-model and the pre-conditions of the model transformation under test class2rdbms. The class on top of the partial model matches class B of the reference model, the class on the left matches class D (both are persistent), the class on the right matches class C (name, is_persistent, attrs are not concerned with this matching), and the association of the partial model matches association assoDC. Then the partial model P3 matches the model M1. We notice that the completed model illustrated in Figure 4 (c) doesn't match the model M1 (both subclasses are persistent for instance).

We apply this iterative process to obtain 14 partial models that when completed give a 100% mutation score. In Table 3, we list the partial models, the reference human-made models they were extracted from, and the objects they contained. We notice that partial models do not need specification of all attribute values. For instance, the partial model P3 (Figure 4) has three classes but is_persistent attribute is set to true for only one of them (true or false or nothing in different partial models). Moreover, thanks to the relaxed metamodel, we do not have to instantiate the root container class ClassModel for the partial model. In the right hand side of Table 3, we describe the elements in the 8 human-made reference models. We do not list the abstract classes *classifier* since it is never instantiated and the property *attrs* which is redundant for all models.

4.4 Test Sets Comparison

The set of 14 partial models is made of 109 elements while the set of reference models contain 231 elements which is more than twice. We express less information in the partial models, considering only testing information based on potential faults simulated in the mutants. We also need to specify fewer meta-classes and properties in partial

Table 4. Summary of Mutation Analysis

Human-made model	Mutation Score of the model	Partial models matching the human-made model	Mutation Score of the completed partial models	Mutant killed by model also killed by completed partial models	Additional mutants killed by the completed partial models (% total #mutants)
M1	57.2%	P2, P3, P13	87.1%	100%	29.9%
M2	57.2%	P1, P12	28.9%	50.5%	0%
M3	49%	P1, P4, P7, P10, P12	74.2%	89.3%	39.7%
M4	57.2%	P1, P8, P11, P12	85.6%	98.2%	29.4%
M5	35.6%	P1, P5, P12	39.7%	100%	4.1%
M6	61.3%	P1, P2, P4, P6, P8, P12, P13, P14	83%	100%	21.6%
M7	47.4%	P1, P9, P12	56.2%	100%	8.8%
M8	58.2%	P1, P12, P14	75.8%	100%	17.5%
8 models	100%	14 partial models	100%	100%	0%

models. Our tool generates complete test models form partial models while satisfying the metamodel specification, well-formedness rules, pre-conditions, and a finite scope. In case a partial model violates a pre-condition the tester is notified such that he/she can modify the partial model to satisfy the constraints. The concise size of a partial model facilitates this process. From these results we deduce that creating partial models is simpler compared to creating complete human-made models hence addressing **Q2**.

In a second experiment, in order to study the resilience in quality (in terms of mutation score) irrespective of solver technology (Alloy and SAT in our case) in our methodology, we generate 100 complete models for 14 partial models giving rise to 1400 test models.

A sample of a completed model is illustrated in Figure 4 (c) for a partial model in Figure 4 (a). The time taken to generate a complete test model was almost instantaneous. Setting up a problem in CNF for a SAT solver takes about 400 ms for our case study. The scope to generate complete models is controlled such that the models are structurally minimal. The integer scope was set to $\pm 2^{10}$, and the number of objects per class was adjusted such that it either equals the number of objects in the partial model or the scope is incremented such that a consistent solution is generated. For instance, all Class objects require a primary Attribute object hence the number of Class and Attribute objects were equal or the number of Attribute objects was higher if required by the partial model.

We compute the mutation score of the human-made models and compare it to the score obtained by completing partial models. The results of mutation analysis are summarized in Table 4. The time taken for mutation analysis using 1400 models to detect 200 mutants was about 6 hours and 1 minute.

In Table 4, Column 1 lists the human-made models and column 3 lists the partial models that match parts of the human-made models. In Column 2, we present the mutation score obtained for a human-made model lets say *Mx*. In Column 4, we present the mutation score obtained by the set of completed partial models that match *Mx*. For

instance, M1 kills 57.2% of all mutants, while its set of completed partial models from P2, P3, and P16 (3*100 completed models) kills 87.1% mutants. The column 5 lists the percentage of the mutants killed by reference human-made model that are also killed by the completed models. In column 6, we also present the mutants additionally killed by the completed partial models that were not killed by their reference human-made model. The decomposition in different lines comparing each human-made model with its partial models illustrates that (i) a partial model can match several models and that (ii) a set of several partial models matching one model is more efficient than it. The most important line is the last one where the results of the entire set of human-made models and the entire set of completed partial models are the same : both 100 %. A general conclusion is that both human-made models and completed partial models give a 100% mutation score hence addressing question **Q1**. An in-depth analysis reveals that for 6 human-made models among 8, more than 98% of the mutants killed by the reference model are also killed by the completed partial models. The completed models from P1, P12, P14 kill 89.3% of the mutants killed by M3. Only completed models from P1, P12 have a lower score of 28.9% compared to the score of M2 viz. 57.2%.

Despite generating different solutions for the same partial model our mutation scores remain consistent and comparable to human-made models. We illustrated that a set of partial models has the same efficiency in revealing bugs than the set of human-made models they match. It is not necessary to write complicated human-made models since partial models have the same efficiency.

5 Related Work

The work in this paper integrates some contributions coming from previous work. In [26] Sen et al. have introduced a tool CARTIER (now called PRAMANA) 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. PRAMANA does not integrate a way to translate existing models to ALLOY since it's thought for model synthesis, and does not have any support for partial models. In this work we are able to apply PRAMANA to model completion by feeding it with a suitable translation of the partial models. The idea of generating test models from partial knowledge developed from our previous work in [28]. In [28], we present the idea of generating suggestions to complete models in a domain-specific model editor. We qualify our approach using our previous work [21], where we extend mutation analysis to MDE by developing mutation operators for model transformation languages.

We explore two main areas of related work : specifying partial models and test model synthesis.

The notion of a partial model in MDE has been previously developed for various objectives. In [14], the authors present the notion of *model fragments* that are essentially partial models with test coverage criteria. Similarly, in [24] the authors propose the notion of *model snippets*. Model snippets are partial models that are specified on a relaxed version of the original metamodel. In this paper, we use the notion of model snippets to define partial models. In [20], the authors propose partial models as a way

to represent uncertainty and variation in an already available complete model. They use ALLOY to generate variable complete models.

The second area of related work is about model generation or using partial knowledge to synthesize models for applications such as testing. Model generation is more general and complex than generating integers, floats, strings, lists, or other standard data structures such as those dealt with in the Korat tool of Chandra et al. [8]. 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 [18] which do not contain domain-specific constraints. Models are complex graphs that must conform to an input meta-model specification, a transformation pre-condition and additional knowledge such as a partial model to help detect bugs. In [9], the authors present an automated generation technique for models that conform only to the class diagram of a metamodel specification. A similar methodology using graph transformation rules is presented in [13]. Generated models in both these approaches do not satisfy the constraints on the metamodel. In [27], we present a method to generate models given partial models by transforming the metamodel 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. Previous work exists in mapping UML to ALLOY for a similar purpose. The tool UML2Alloy [2] takes as input UML class models with OCL constraints. The authors present a set of mappings between OCL collection operations and their ALLOY equivalents.

6 Conclusion

Manually creating test models that conform to heterogeneous sources of knowledge such as a metamodel specification, well-formedness rules, and testing information is a tedious and error-prone operation. Moreover the developed test models are often unreadable, since their complexity obfuscates the testing intent.

In this paper, we present methodology based on models from partial knowledge to simplify the development of effective test models. The emphasis in this paper is to push towards the development of partial models by analyzing requirements, existing test models [7], the transformation under test [16], or fault locations [12]. We provide a semi-automated tool [25] to support the development of partial models and completing them by automatic solving. We show that the specification of partial models is concise, they can detect the same bugs that a human-made model can detect, and they precise capture testing intent. We also perform experiments show that partial models are effective irrespective of the underlying solver (which is Alloy in this case). Different non-isomorphic complete models obtained by solving a single partial model consistently detect the bug the partial model was initially targeted to kill.

We believe our work reinforces a human-in-the-loop testing strategy that sits in between two prominent schools of thought: manual test case specification and automated test generation. Partial models are an effective way to insert human testing knowledge or sophisticated analysis in a purely automated generation methodology. The approach presented in the paper contains steps that are automated and those that are not. For instance, the transformation of a partial model to ALLOY could be automated to a certain degree. The scoping strategy can be improved using various heuristics to synthesize a diverse set of complete models. We would also like to explore strategies to combine mutually consistent partial models to generate a smaller set of complete test models. Finally, we would like to see the creation of industry-strength tools that allow convenient ways to specify partial models with fully automated complete test model synthesis.

References

1. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. Uml2alloy: A challenging model transformation. In *MoDELS*, pages 436–450, 2007.
2. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On Challenges of Model Transformation from UML to Alloy. *Software and Systems Modeling, Special Issue on MoDELS 2007*, 9(1):69–86, 2008.
3. V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser. Traceability mechanism for error localization in model transformation.
4. R. Bardohl, G. Taentzer, and A. S. M. Minas. *Handbook of Graph Grammars and Computing by Graph transformation, vII: Applications, Languages and Tools*. World Scientific, 1999.
5. B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu. Barriers to Systematic Model Transformation Testing. *Communications of the ACM*, 53(6), 2010.
6. 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.
7. A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Modeling Model Slicers. In *ACM/IEEE 14th International Conference MoDELS*, volume 6981, pages 62–76, Wellington, Nouvelle-Zélande, Oct. 2011. Springer Berlin / Heidelberg.
8. 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.
9. 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.
10. F. Budinsky. *Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2004.
11. DeMillo, R. L. R., and F. Sayward. Hints on test data selection : Help for the practicing programmer. *IEEE Computer*, 11(4):34 – 41, 1978.
12. P. Dhoolia, S. Mani, V. Sinha, and S. Sinha. *Debugging Model-Transformation Failures Using Dynamic Tainting*, volume 6183, pages 26–51–51. Springer, 2010.
13. K. Ehrig, J. Kster, 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.
14. 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.
15. D. Jackson. http://alloy.mit.edu. 2008.
16. C. Jeanneret, M. Glinz, and B. Baudry. Estimating footprints of model operations. In *International Conference on Software Engineering (ICSE'11)*, Honolulu, USA, May 2011. IEEE.

17. F. Jouault and I. Kurtev. On the Architectural Alignment of ATL and QVT. In *Proceedings of ACM Symposium on Applied Computing (SAC 06)*, Dijon, FRA, April 2006.
18. H. M and J. Power. 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.
19. Y. S. Mahajan and S. M. Z. Fu. Zchaff2004: An efficient sat solver. In *Lecture Notes in Computer Science SAT 2004 Special Volume LNCS 3542.*, pages 360–375, 2004.
20. M. C. R. S. Michalis Famelis, Shoham Ben-David. Partial models: A position paper. In *MoDeVVa Workshop (co-located with MODELS 2011)*, 2011.
21. J.-M. Mottu, B. Baudry, and Y. L. Traon. Mutation analysis testing for model transformations. In *Proceedings of ECMDA'06*, Bilbao, Spain, July 2006.
22. P. A. Muller, F. Fleurey, and J. M. Jezequel. Weaving executability into object-oriented meta-languages. In *Inernational Conference on Model Driven Engineering Languages and Systems (MoDelS/UML)*, pages 264–278, Montego Bay, Jamaica, 2005. Springer.
23. OMG. The Object Constraint Language Specification 2.0, OMG: ad/03-01-07, 2007.
24. R. Ramos, O. Barais, and J.-M. Jzquel. Matching model-snippets. In *MoDELS*, pages 121–135, 2007.
25. S. Sen. Partial model completion tool. https://sites.google.com/site/partialmodeltool/, 2011.
26. S. Sen, B. Baudry, and J.-M. Mottu. On combining multi-formalism knowledge to select test models for model transformation testing. In *IEEE International Conference on Software Testing*, Lillehammer, Norway, April 2008.
27. 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.
28. S. Sen, B. Baudry, and H. Vangheluwe. Towards Domain-specific Model Editors with Automatic Model Completion . *SIMULATION*, 86(2):109–126, 2010.
29. E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for Construction and Analysis of Systems*, Braga,Portugal, March 2007.