Searching the boundaries of a modeling space to test metamodels

Juan Cadavid, Benoit Baudry, Houari Sahraoui

Searching the boundaries of a modeling space to test metamodels

Juan José Cadavid Gómez
INRIA, Centre Rennes
Bretagne Atlantique, Rennes, France
Email: juan.cadavid@inria.fr

Benoit Baudry
INRIA, Centre Rennes
Bretagne Atlantique, Rennes, France
Email: benoit.baudry@inria.fr

Houari Sahraoui
Université de Montréal
Informatique et Recherche Opérationnelle
Montréal, Québec, Canada
Email: sahraouh@iro.umontreal.ca

***Abstract*—Model-driven software development relies on metamodels to formally capture modeling spaces. Metamodels specify concepts and relationships between them in order to represent either a specific business domain model or the input and output domains for operations on models (*e.g.*, model refinement). In all cases, a metamodel is a finite description of a possibly infinite set of models, *i.e.* the set of all models which structure conforms to the description specified in the metamodel. However, there is currently no systematic method to test that a metamodel captures all the correct models of the domain and no more. In this paper, we focus on the automatic selection of a set of models in the modeling space captured by a metamodel. The selected set should both cover as many representative situations as possible and be kept small as possible for further manual analysis. We use simulated annealing to select a set of models that satisfies those two objectives and report on results using two metamodels from two different domains.**

I. Introduction

A metamodel is a set of concepts within a domain of knowledge and the possible relationships that may occur between them. As such, a metamodel defines the structural properties that must be satisfied by all models in that domain. For example, the UML metamodel [1] defines the structure of models that represent software systems, while the SPEM metamodel [2] defines the structure of process models. More generally, the construction of a metamodel aims at formally capitalizing the structure of a domain in order to let all the stakeholders in that domain exchange models that share structural constraints. Metamodels also serve as a basis to build model editors, model analyzers and code generators for a specific domain.

Although metamodels play a central role for the definition of a domain and all associated tools, metamodeling is still a craft, where the craftsmen are the domain experts. They usually look at existing practices, exchange with stakeholders who build models in that domain and identify the key concepts that are necessary to describe abstractions in that domain. Then, they refine this list of concepts, add attributes and relationships, and this provides an initial metamodel. A major issue when designing a metamodel is usually to set the boundaries correctly, *i.e.* set all relationships and multiplicity constraints correctly. However, there is no systematic technique to test these boundaries correctly.

In this paper we propose an automated technique to sample the modeling space captured by a metamodel. This consists in the automatic generation of a set of *test models* that conform to the constraints defined by the metamodel and that globally cover the modeling space. A set of test models will enable domain experts to confirm possible desired situations. But more importantly, it will assist them in identifying undesired situations, pinpointing faults in the metamodel, leading to corrections to obtain a *precise metamodel*. It becomes thus necessary to find an adequate, manageable set of test models, allowing the experts to test a metamodel.

An adequate set of test models should cover the modeling space, showing all relevant cases of possible models, and should favor diversity to increase the chances of revealing different findings. The coverage of the modeling space is based on previous work by Fleurey et al. [3], which proposes the notion of model fragments to partition the space. We define a new diversity function for a set of models, which evaluates whether the models in the set cover different parts of the space. Both coverage and diversity are formalized and passed as a fitness function for a simulated annealing search that aims at automating the selection of an adequate set of test models.

A series of experiments with two metamodels demonstrates that (i) our approach can automatically provide the expert with a diverse set of models that covers the modeling space; (ii) our choice of parameters for simulated annealing search represents the best choice in maximizing these criteria; and (iii) our approach systematically generates a better set than random search with respect to coverage and diversity.

The paper is organized as follows. Section II explores the metamodeling definitions necessary to state our problem, which is presented in section III. Section IV discusses our proposed approach and the criteria used to qualify models. Section V discusses our approach based on the search-based technique of Simulated Annealing. In section VI we present our empirical validation study of our approach, section VII presents related work and section VIII points out our conclusions and future work.

II. Background

In this section, we present the basic definitions to describe our problem. Then, we illustrate our motivation with examples.

A. Metamodeling

A *metamodel* defines the abstract syntax of a modeling language. In the same way a formal grammar provides production rules for a formal language, a metamodel specifies what kind of constructs are available in a modeling language and how these constructs can relate to each other. Normally, modeling languages also provide a concrete syntax to allow the user visualize models in a friendly way. We present a simplified definition of metamodel presented in the Meta-Object Facility (MOF) specification [4].

Definition 1: **Metamodel (MM)**. A metamodel is defined as the composition of:

- **Classes**. The core concepts and attributes that define the domain of the modeling language. Clases may contain properties, specifying the range of values each one can take.
- **Relationships**. Associations between classes that specify how the concepts can be bound together in this modeling language. Relationships define multiplicities, specifying the minimal and maximal number of occurrences.

When a model is built with a particular modeling language, we say that this model is an *instance* of the metamodel of this modeling language. Formally, the relationship between models and metamodels is given by the *instanceOf* predicate [5].

Definition 2: **instanceOf(m, MM)**. A model instance m of a metamodel MM is such that:

- Every object o in m is the instance of a class C in MM. It is also said that o is of type C.
- Every link between two objects in m is such that it exists, in MM, a relationship between the two classes typing the two objects.
- Every semantic property defined in MM is satisfied in m. For instance, the multiplicity defined on references between concepts denotes a range of possible links between objects of these classes (*i.e.* concepts).

Example 1: Figure 1 provides a subset of the metamodel for Feature Diagrams, a modeling language issued from the field of Feature-Oriented Domain Analysis [6]. It was created to allow software developers to reason over a large number of variants for their software systems, composed of different sets of features. The notation used in the figure is provided by the MOF standard. A `Feature` represents a functional feature of a product. A `Feature` has a name and may or may not be optional. It may have zero or more children `Features`, as specified in the multiplicity "0..*" in this relationship in the metamodel, meaning that this feature is the composition of several simpler features (also known as subfeatures). The parent relationship is the opposite of the children relationship. A special type of `Feature` is `PrimitiveFeature`, which are elemental features that cannot be further decomposed and thus have no subfeatures. `Features` can also relate to each other through `Operators`. An `Operator` is owned by a `Feature` and might have one or more `Features` as operands, as specified in the multiplicity "1..*". The `And` operator means that all of its operand `Features` must be present in a product, whereas the `Or` operator means that at least one of them must be present. Figure 2 shows a feature diagram drawn with the concrete syntax of the modeling language; `Features` are represented as rectangles and the `Or` operator is represented as a filled circular sector. The model defines the Gearbox feature which can have as subfeatures Manual or Automatic transmission. Figure 3 shows the same model, otherwise represented in the notation of UML object diagrams, in order to represent in an explicit way how this model contains object instances of classes in the metamodel as well as the value of each one of its properties.

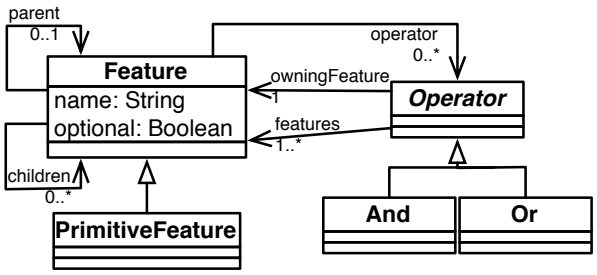

Figure 1. Metamodel for Feature Diagrams.

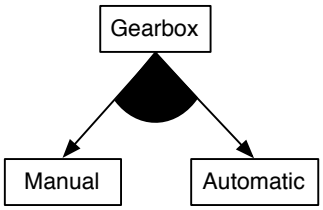

Figure 2. A feature diagram in the modeling language's concrete syntax.

B. Motivating metamodel testing

Although MOF is a powerful yet simple language to define metamodels, their construction can be difficult and is prone to many possible faults. To illustrate the kind of errors that can occur in a metamodel, and the way these errors can be exhibited by test models, we examine once again the metamodel for Feature Diagrams.

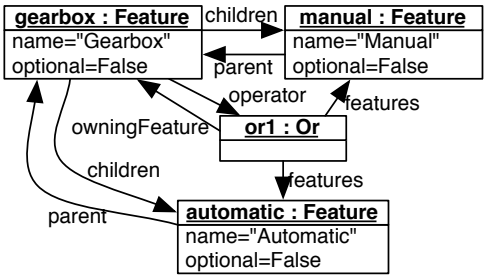

Figure 3. Object diagram for a Feature Diagram model instance.

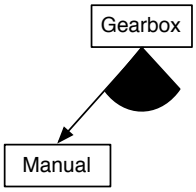

Figure 4. Incomplete feature diagram; the Or operator has only one operand.

Example 2: Figure 4 shows again the example of a feature diagram, this time showing only one subfeature for Gearbox. The feature diagram in this example portrays the configuration of a product that presents the Gearbox `Feature`, which has as a child, the Manual `Feature`. The two features are linked by an `Or` operator. This model satisfies the abstract syntax of the metamodel presented in figure 1. However, to the domain expert of feature diagrams, it reveals an error in the metamodel. The `Or` operator has only one operand; the logical disjunction operator implies there should be at least two operands. The concerned relationship of the metamodel is the "features" relationship, originating from the `Operator` metaclass. It has been defined with a multiplicity [1..*], effectively allowing the situation of an `Or` operator with only one operand to take place. In this case, we say the metamodel is *underconstrained*, because it allows for undesired situations to occur. In order to alleviate this problem, the expert can correct the metamodel by increasing the lower bound of the multiplicity range to 2.

Example 3: Let us look again at the feature diagram on figure 2. It expresses that the feature Gearbox can have Automatic *or* Manual transmission. However, this operator refers to the logical operation of inclusive disjunction, in which *one or more* of the operands are required to be present, therefore indicating that a Gearbox may be both Automatic and Manual. As this is not the case according to the domain, a new operator is needed to represent the disjunctive nature of these features. By providing only the `Or` operator, the metamodel is specifying that all models containing operators to link features specify either the inclusive disjunction operator `Or` or the conjunction operator `And`. In this form, it is refraining models from expressing any other operator than these two. We say in this case that the metamodel is *overconstrained*. To remedy this problem, a third operator which accurately captures the relationship between features, `Xor`, is added to the metamodel.

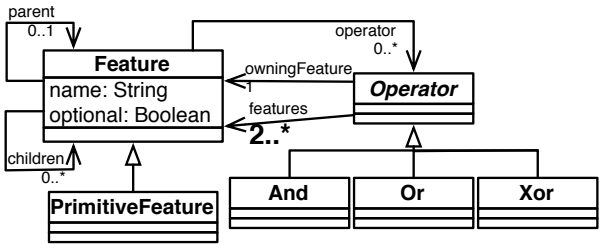

Figure 5. Metamodel for Feature Diagrams with the corrected multiplicity and the Xor operator.

Figure 5 shows the metamodel with the two mentioned corrections.

Our approach aims at providing the expert with a set of models that will allow him to test the metamodel, by spotting errors in the generated sets of models. To test a metamodel, the only testing oracle is the expert, who in his knowledge of the domain will determine what makes a correct instance or not. As there is no way to possibly automate this, a small, manageable and test-adequate set of models will allow him to test his metamodel, and make adjustments as needed in order to obtain a *precise metamodel*.

III. Problem Statement

In what follows, we establish the formal definition for our problem. A metamodel captures the set of all the possible valid models for a modeling language. We call this set the *modeling space*.

Definition 3: A **Modeling Space (MS)** is the set of all possible model instances that conform to a metamodel:

$$MS(MM) = \{m : model | \text{instanceOf}(m, MM)\}$$

As illustrated in figure 6, the modeling space captured by a metamodel might be larger than initially intended by the domain expert. As shown in the preceding examples, there are models that satisfy the abstract syntax defined by a metamodel, yet they are incorrect according to the domain's expert and thus do not belong to the *intended modeling space*. This gap between the default modeling space captured by a metamodel and the precise modeling space intended by the expert, leaves room for incorrect models, thus indicating faults in the metamodel. Thus, the metamodel must be tested to find undesirable situations to remove from the captured modeling space, as well as adding missing desired situations. In section II, example 2, a detected undesired situation resulted in a correction that contracted the boundaries of

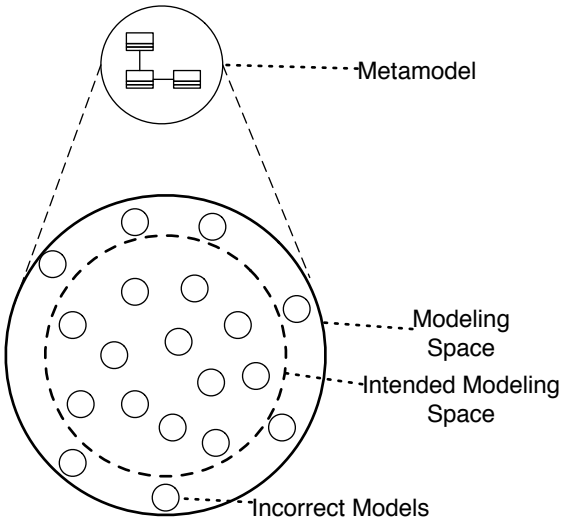

Figure 6. Default modeling space created by a metamodel and precise modeling space

the modeling space to include only correct multiplicities. On the other hand, example 3 showed how the metamodel being overconstrained resulted in a modification which expanded the boundaries to contain instances with a new operator.

In an industrial context, a more critical outcome is visible as experts across different domains design modeling languages with metamodels which tend to be much larger, defining a significant higher number of classes and relationships. Therefore, an automated approach aimed at obtaining data to test such metamodels becomes necessary. The following section presents our approach to solve this problem, leveraging search-based techniques to avoid the combinatorial explosion resulting from the exploration of the entire modeling space.

IV. Criteria for the Qualification of Test Models

In this section we introduce the generalities of our approach. Then, we present the mechanism to qualify the test models and the formal definitions to quantify the satisfaction of the criteria by a set of test models.

A. Approach

In a nutshell, our approach starts from a metamodel MM and selects a sample from the modeling space captured by MM. In order to provide a relevant sample to test the metamodel, such sample must satisfy the following characteristics:

- Small, to allow the expert to detect faults in the metamodel with a reduced effort in manual analysis.
- Complete, to cover all possible concepts, relationships and property values defined in a metamodel.
- Dissimilar, *i.e.* the models within the sample should bear little similarity between themselves, in order to increase diversity.

The goal of making the set of test models small is parallel to that of making it dissimilar; a smaller set will contain less similarities between the models than a larger set. Therefore, our approach focuses in the maximization of *coverage* and *dissimilarity*, and as such it guarantees the coverage of relevant cases of the modeling space and favors diversity to increase the chances to reveal different findings. The idea of defining a function to evaluate for similarity among test cases in order to increase diversity has been treated before in other software testing approaches [7]. In this section we provide formal definitions to measure both criteria. However, before that we examine the underlying framework that permits us to measure these criteria in models, according to the modeling space captured by its metamodel.

B. Model Fragments to Qualify Test Models

The qualification of model instances according to adequacy criteria is based on the mechanism presented in [3], which aimed at qualifying test data for model transformations. This approach is based on category partition testing, which guarantees input domain coverage. It creates a partition in every field in a metamodel; therefore making the total set of such partitions the complete set of possible value settings of the metamodel. This approach defines the notion of Model Fragments. A model fragment is the specification of a range of values of the metamodel that a model instance may cover or not.

Definition 4: A **Model Fragment (MF)** of a metamodel MM is a tuple of:

- A class of the metamodel MM.
- The property or relationship of the class for which it specifies a value or a multiplicity range.
- The range of values for such property.

A model fragment is covered by a model if such model contains an object with a property in the declared value ranges of the model fragment. An exhaustive definition can be found at [3].

Example 4: Model fragments computed for the metamodel in figure 5.

```
1: mf(Feature,name,StringRange
     (regexp: ))
2: mf(Feature,name,StringRange
     (regexp: +))
3: mf(Feature,optional,BooleanRange
     (boolValue: true))
4: mf(Feature,optional,BooleanRange
     (boolValue: false))
5: mf(Feature,operator,IntRange
     (lower: 0, upper: 0))
```

```
6: mf(Feature,operator,IntRange
    (lower: 1, upper: 1))
7: mf(Feature,operator,IntRange
    (lower: 2, upper: MAX_INT))
8: mf(Feature,children,IntRange
    (lower: 0, upper: 0))
9: mf(Feature,children,IntRange
    (lower: 1, upper: 1))
10: mf(Feature,children,IntRange
    (lower: 2, upper: MAX_INT))
11: mf(Feature,parent,IntRange
    (lower: 0, upper: 0))
12: mf(Feature,parent,IntRange
    (lower: 1, upper: 1))
13: mf(Operator,owningFeature,IntRange
    (lower: 1, upper: 1))
14: mf(Operator,features,IntRange
    (lower: 2, upper: MAX_INT))
```

It can be seen that the overall set of 14 fragments form the complete partition of ranges of all properties and relationships defined in the metamodel. For the "name" property of `Feature`, there are fragments specifying both the empty ("") and non-empty string ("+"). For the "optional" feature, fragments are generated with both possible values, true and false. For the relationships, their multiplicity is fragmented according to the allowed multiplicities as they were specified in the metamodel. For example, in the case of the "children" relationship, which has a multiplicity of 0..*, a partition such as $\{\{0\}, \{1\}, \{x|x \leq 2\}\}$ is defined to ensure that the partition of fragments cover the occurences of this property with zero, one and more than one objects.

For our purposes we define the covering function, a boolean function that determines whether a model covers or not a model fragment.

Definition 5: The function $covering(MF, MI)$ returns true if:

- If the model instance (MI) contains at least an object for which the model fragment (MF) provides a range of values.
- If such object defines a property within the value or multiplicity range that the model fragment (MF) specifies.

Example 5: The instance given in figure 2 covers the fragments 2, 4, 5, 6, 8, 10, 11, 12, 13 and 14 of the Feature Diagram metamodel.

Finally, we define the total set of model fragments computed for a metamodel as follows.

Definition 6: A **Metamodel Fragments Partition (MFP)** is the set of all possible model fragments for a metamodel, where there is a model fragment for each value and multiplicity range for each property of each class in the metamodel.

In order to compute model fragments for any given metamodel, our approach leverages on Metamodel Coverage Checker [3], which automatically calculates the MFP for an input metamodel.

C. Criteria Definitions

For the following definitions, let sm be a set of models such that $sm \subseteq MS(MM)$. Let $CMF(sm)$ be a function that returns the set of model fragments covered by a sm; in other words, every model fragment being covered by at least one model.

$$CMF(sm) = \{mf \in MFP | \exists mi \in sm | \\ covering(mf, mi)\}$$

1) Metamodel coverage criterion: The portion of modeling space covered by a set of test models is measured by the number of model fragments the set covers, normalized by the size of the entire model fragment partition.

$$coverage(sm) = \frac{|CMF(sm)|}{|MFP|}$$

2) Dissimilarity criterion: As a small set will be sustained, different test models showing heterogeneous scenarios will allow to explore different cases and reveal new findings. We quantify the diversity of the set of test models by counting the number of fragments which are covered more than once by the test models in the set. Let $IC(mf)$ be a function that returns the subset of instances covering a specific model fragment.

$$IC(mf) = \{mi \in MS | covering(mf, mi)\}$$

We calculate excess coverings (EC), the sum of the number of times every model fragment has been covered in excess, *i.e.* more than once.

$$EC = \sum_{i=1}^{|CMF|} |IC(mf_i)| - 1$$

In order to normalize this count of excess coverings, we have defined a metamodel-specific parameter called model fragment redundancy tolerance (MFRT). This parameter specifies the number of times we will tolerate a model fragment to be covered in excess; we have set its value to one tenth of the size of the model fragments partition, which allows for a tolerable level of similarity between instances.

$$MFRT = \frac{|MFP|}{10}$$

The dissimilarity of the set of models is thus computed as follows.

$$dissimilarity(sm) = max\left(0, 1 - \frac{EC}{MFRT \times |CMF(sm)|}\right)$$

The following section presents how we have implemented our approach, while taking these two definitions to evaluate the adequateness of the set of test models.

V. Solution Search Strategy

To achieve finding the best possible set of test models in a modeling space that best satisfies our criteria, it would be required to perform a complete combinatorial coverage. As this is prohibitively expensive computationally, we have leveraged search-based techniques to build our approach, being able to explore the search space. In our case, the search space are all the possible combinations of models contained in the modeling space. This section explains the implementation of our approach with our chosen search-based technique.

Simulated Annealing (SA) is a local search metaheuristic inspired by the metal annealing process of metallurgy, where a crystalline solid is heated and then allowed to cool down according to a cooling schedule until it achieves its most regular possible crystal configuration, and thus is free of crystal defects. To succeed in this, the lattice energy state is minimized, thus obtaining a homogeneous material structure [8]. An interesting quality of this technique is that the evaluation of the fitness of the solution occurs only once per iteration, and therefore it is appropriate for applications where the fitness function deals with complex structures, as it is our case. We have chosen this technique for this reason.

It is important to remember nevertheless, that the motivation for search-based techniques is that applications do not require strictly finding the global optimum, but a "good" solution would suffice to achieve the application-specific goals. For example, in generation and selection of test data for programs, the goal is to cover more of the uncovered paths; however an almost perfect solution can satisfy the needs of testers, as suggested in surveys of search-based approaches in software engineering such as [9] and [10].

Algorithm 1 shows the Pseudocode of the implemented Simulated Annealing algorithm. As it is a local search technique, it works in a single thread of execution while creating and evaluating only one solution at every iteration. The basic workflow is as follows. The initial temperature is set. An initial random solution is taken from the search space. Its fitness value is calculated. Then, the iterative cycle of the algorithm begins. This cycle is controlled by the temperature parameter, which ends once the temperature is cold enough *i.e.* the minimal temperature has been reached. In every iteration, the neighborhood function is invoked to obtain a new solution, very similar to the current solution. The fitness value for this new solution is computed, and if it is higher to that of the current solution, the new solution is accepted as the new best. If it is not higher, the solution can still be accepted according to a probability function (this is the mechanism that prevents the algorithm from getting trapped in local optima). The current temperature is decreased according to the cooling schedule, and a new iteration begins.

Algorithm 1 Simulated Annealing simplified pseudocode.

```
1:  t ← INIT_TEMPERATURE
2:  s ← INIT_SOLUTION
3:  e ← measure(s)
4:  sbest ← s {set best solution to initial solution}
5:  while t < MIN_TEMPERATURE do
6:    for i = 1 → N_ITERATIONS do
7:      snew ← neighborhood(s) {explore new solution
        in the space}
8:      enew ← measure(snew) {fitness function}
9:      Δs ← e − enew
10:     if Δe < 0 then
11:       s ← snew {better solution, accepted}
12:       e ← enew
13:       if enew > ebest then
14:         sbest ← snew {new best solution}
15:         ebest ← enew
16:       end if
17:     else
18:       random ← random()
19:       probability ← exp(−Δe/t)
20:       if random < probability then
21:         s ← snew {bad solution accepted}
22:       end if
23:     end if
24:   end for
25:   t ← coolDown(t)
26: end while
27: return sbest.
```

A. Neighborhood Function and Modeling Space Generation

In every iteration, the SA algorithm explores the search space by evaluating a solution very similar to the current one. In order to find 'nearby' solutions, the SA algorithm requires the definition of a neighborhood function. This function, applied in line 7 of the shown pseudocode, takes as input the current solution and returns a new solution, which should be similar enough to cause a slight variation on the fitness value. Our implementation of this function takes the current solution as an input (i.e. the current set of models) and applies randomly one of two possible operations: removes a random test model from the current set in evaluation or adds a random test model from the modeling space.

On the other hand, it is also requirement for a neighborhood function to be able to obtain any possible solution within the search space. We use Alloy [11] to enable the generation of the modeling space. Alloy is lightweight formal language to specify object models and its associated well-formedness rules as basic first-order logic formulas. Its accompanying tool, Alloy Analyzer, permits to generate automatically random examples for object models. This capacity has allowed several Model-Driven Engineering ap-

proaches to take advantage for the automatic generation of test instances for metamodels [12], [13]. We have created an Alloy-powered Java program capable of enumerating the complete modeling space for the input metamodel.

B. Fitness function

Perhaps the most critical aspect of the implementation of a search-based technique, is the definition of a function that allows the algorithm to tell a better solution from another (line 8). In order to balance out the criteria given in section IV, we have developed two versions of the fitness function. For a specific set of models sm, its fitness value is calculated as follows.

1) Weighted fitness function: This version of the fitness function aims to assign a weight to each one of the two adequateness criteria. The α value will determine the weight given to the criteria of metamodel coverage.

$$\begin{aligned} score(sm) = \quad & \alpha * coverage(sm) \\ & +(1-\alpha) * dissimilarity(sm) \end{aligned}$$

2) Minimal Coverage Multi-category fitness function: It aims to guarantee a minimal coverage of the metamodel, and have solutions not meeting such coverage be penalized. This causes the SA algorithm to reject them. We believe that in most cases where model instances will be used for testing in the scenarios exposed in section II, the minimal coverage should be 90%. Thus the $MIN\_COVERAGE$ parameter is set to 0.9.

$$score(sm) = \begin{cases} dissimilarity(sm)/2 \\ \text{if } coverage(sm) < MIN\_COVERAGE \\ 0.5 + dissimilarity(sm)/2 \\ \text{if } coverage(sm) \geq MIN\_COVERAGE \end{cases}$$

C. Additional details of implementation

Below are other important implementation decisions.

Numeric parameters: The outcome of search-based techniques such as simulated annealing is highly sensitive to the choice of parameters that control its execution. In the implementation of our approach, we found that the values of the numeric parameters (initial and final temperature, cooling factor, number of iterations) determinate the quality of the resulting solutions, with respect to the defined test-adequacy criteria in section IV. We analyze this behavior in our validation experiments.

Initial solution: In our case, the initial random solution picked by the algorithm to start iterating corresponds to a random set of models from the modeling space of the metamodel under test. The size of this set is also random.

Probability function: We use the standardized form for SA implementations which follow the Metropolis-Hastings algorithm [14]. The decision to move to a new solution is not ruled by the fact that the new solution has to be a better one, but a probability function is provided to allow for bad solutions to be accepted while the temperature is high.

Implementation in Code: The approach was built in Java. In order to manage the complexity of models, the Eclipse Modeling Framework (EMF)[15] was used. The Metamodel Coverage Checker, also implemented in Java, analyzes the structure of the metamodel in order to construct the partition of model fragments. The metamodel is encoded in Ecore, which is EMF's dialect of the MOF standard for the construction of metamodels. Model instances are encoded in XMI (XML Metadata Interchange).

VI. Validation

The *goal* of our empirical study is to validate the configuration of parameters and effectiveness of our approach. The *quality focus* is the ability of our approach to generate sets of models that effectively allow the expert to detect faults in the metamodel by pinpointing the boundaries of the modeling space described by the metamodel. The *perspective* to be confirmed is that our approach, successfully applied to two metamodels, one of them at the scale of an industrial application, can be applied to other domains. The research questions we aim to answer with this empirical study are:

- Does our approach provide automatically sets of test models for a given metamodel, satisfying the defined criteria coverage?
- What is the configuration of parameters for our approach, that yields the best results, judged by the measure of the test adequacy criteria?
- Is our approach to generation of sets of test models better than random generation judged by the measure of the test adequacy criteria? If so, to what extent?

A. Metamodels for Experimental Data

We have taken two metamodels at different scales.

- **Statecharts Metamodel.** A reduced version of the statecharts metamodel, containing in total 5 classes, 4 properties and 10 relationships.
- **Feature Diagrams Metamodel.** The complete version of the feature diagram metamodel, as it is formally defined by Perrouin et al. [16]. It contains in total 18 classes, 11 properties and 19 relationships.

B. Parameter configuration

One of the critical aspects of the application of search-based techniques is finding the right configuration of parameters. We sampled the set of all parameter values combinations (section V) by selecting all pair-wise interactions between values, resulting on the 15 configurations shown in table I. Given the importance of metamodel coverage,

Table I
VALIDATION EXPERIMENTS WITH PAIR-WISE CONFIGURATION OF PARAMETERS

Exp ID	Fitness Function	Number of Iter.	Initial Temp.	Final Temp.	Cooling Factor
exp01	Weighted	100	10	0,01	0,9
exp02	Weighted	500	50	0,001	0,9
exp03	Weighted	1000	100	0,0001	0,9
exp04	MinCoverage	100	50	0,0001	0,9
exp05	MinCoverage	500	100	0,01	0,9
exp06	MinCoverage	1000	10	0,001	0,9
exp07	Weighted	100	100	0,001	0,9
exp08	Weighted	500	10	0,0001	0,9
exp09	Weighted	1000	50	0,01	0,9
exp10	Weighted	1000	100	0,0001	0,95
exp11	Weighted	100	10	0,01	0,95
exp12	MinCoverage	500	50	0,001	0,95
exp13	MinCoverage	500	50	0,001	0,99
exp14	Weighted	1000	100	0,0001	0,99
exp15	Weighted	100	10	0,01	0,99

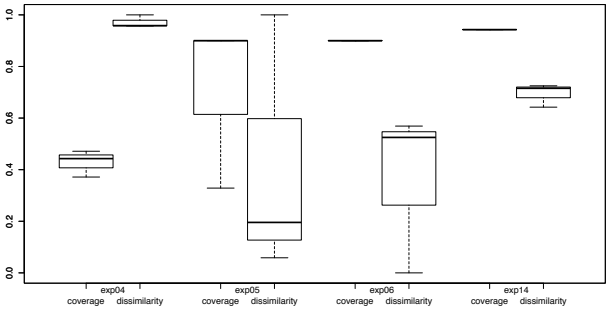

Figure 7. Coverage and dissimilarity measures for the feature diagrams metamodel

we have assigned a weight of 70% of the weighted fitness function. Therefore, the dissimilarity criterion has a weight of 30%, in order to exclude the cases where boundary models (*e.g.* a model with no objects at all) provoke a high dissimilarity measure but do not increase the overall metamodel coverage. Thus the weighting parameter α or the weighted fitness function has been set to 0.7. Throughout the different configurations, we have realized that this value balances out properly the coverage of the adequacy criteria.

Each one of these configurations was executed in series of 3 repetitions each. For reasons of space, we only show the outcome of experiments 4, 5, 6 and 14. The results of the remaining experiments are very similar to these. Figures 7 and 8 show the measures of adequacy criteria for the feature diagram and statecharts metamodels. In both cases, the coverage measure approaches 1, while the dissimilarity measure shows a median of 0.7 for feature diagrams and 0.8 for statecharts. We observe that of all explored configurations of parameters, in both metamodels exp14 shows the best trade-off between dissimilarity and

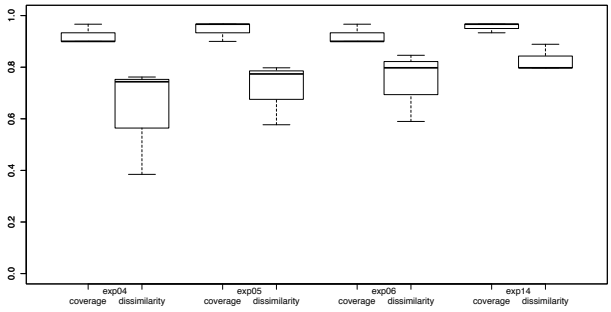

Figure 8. Coverage and dissimilarity measures for the statecharts metamodel

coverage, which suggest that this particular configuration can be applied to execute our approach with other metamodels with equally effective results.

C. Validation against random

To validate the adequacy of the sets of test models generated by our approach against randomly generated sets, we compare the measures of the adequacy criteria in both cases. We use the exp14 configuration of parameter values for this comparison experiment, since this was the most effective in the preceding phase of the empirical validation.

For each metamodel, a series of 20 sets of models were randomly chosen from the modeling space. The size of these sets were also chosen randomly. On the other hand, the implementation of our approach was executed in 20 repetitions, to obtain 20 optimized sets of models. We use the measures of our adequacy criteria (coverage and dissimilarity), to compare both collections of sets.

Figure 9 shows the comparison of the coverage criteria of the random sets versus the sets generated by our approach. As expected, our approach generates sets of test models that effectively cover almost the totality of the metamodel elements and value ranges, against an average coverage provided by randomly generated test sets of 80% for the statecharts metamodel and 50% for the feature diagrams metamodel.

As for the criterion of dissimilarity, figure 10 shows the same comparison. We observe that for the statecharts metamodel, the solution with our approach is significantly more dissimilar than the randomly generated solution. As for the metamodel of feature diagrams, the dissimilarity is improved but to a lesser extent, a median difference of 0.15. This is due to the relative large size of the feature diagram metamodel, which makes it harder to ensure diversity in the test models while keeping a high coverage. Additionally, the dissimilarity scores present a high variation measure, as can be observed by the size of the boxes. This can be due to the weight of 30% this criterion has been assigned to in the

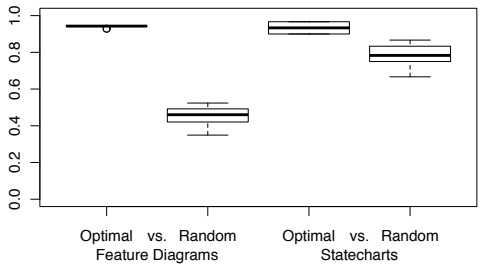

Figure 9. Comparison of the coverage criteria measure, optimal solutions generated by our approach vs. random solutions for both metamodels

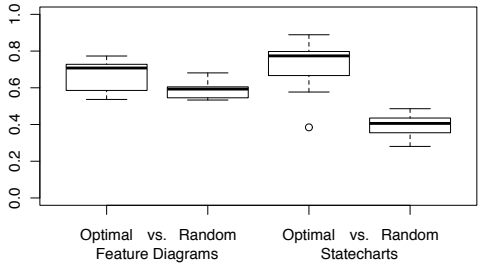

Figure 10. Comparison of the dissimilarity criteria measure, optimal solutions generated by our approach vs. random solutions for both metamodels

implementation of the weighted fitness function.

To this end, we can assert that our approach does improve the results of randomly generated sets of test models, judged by the measure of the adequacy criteria.

D. Conclusions

Our approach effectively provides an optimized set of test models for both metamodels tested. In both cases, a specific configuration of parameters was found, which guarantees an almost maximum of coverage, suggesting that the obtained solutions cover almost 100% of the modeling space captured by both metamodels. Regarding the dissimilarity measure, the scores ranged from 0.7 to 0.8, ensuring the diversity of the solution. The sets of models generated by our approach excel those generated randomly based on the satisfaction of our adequacy criteria.

E. Threats to validity

Despite the results showing that the aimed goals have been attained, as with every empirical validation effort, it is important to acknowledge possible threats to the validity of our experiments. Their nature can be internal, external and of construction.

Construction threats refer to the implementation choices for our approach, as they were explained in section V. As stated before, different possible configurations of parameters for the algorithm were explored, however other details in the implementation such as the neighborhood function, the choice of the redundancy tolerance, of minimal coverage ratio, the fitness functions, the probability function, and the weight in weighted fitness function could influence the overall outcome of our approach. Also, our mechanism to generate the modeling space for a metamodel is assumed to generate the complete modeling space, however we cannot prove that Alloy generates exhaustively all possible instances. This can be relieved by investigating other forms of generation of the complete modeling space within a delimited scope.

Internal validation threats lie on the source of the validation data. As we have experimented with 15 possible configurations of parameters, and chose the best one to perform validation against random generation, it is possible that many other configurations can yield different results. As for the validation against random, other measures beyond the satisfaction of our test adequacy criteria may show different outcomes. Internal threats also lie on possible bugs in the components upon our approach is built, namely Alloy and Metamodel Coverage Checker.

External threats lie on the statistical significance of our study. We acknowledge that we have only observed the behavior of our approach with two metamodels, and thus we cannot guarantee its performance on other metamodels.

VII. Related Work

Different approaches have approached the model instance generation problem. In this section we examine some of them.

1) Precise Metamodeling: The premise of precise metamodels has been treated as early as 2002 when the Meta-Modeling Language (MML) [17] was proposed to extract a subset of the UML 1.3 specification to create static object-oriented models, which would later become one of the bases for MOF. Gogolla et al. [18] provide a mechanism and a tool called USE to validate UML models and OCL specifications by generation of model instances as object diagrams, called *snapshots*, which are generated from a specification of the desired properties to test in the purpose-specific language ASSL. Although not expressly built for metamodel testing, metamodels can also be expressed in UML. Sadilek et al. [19] discuss the necessity to test metamodels annotated with well-formedness rules. It portrays a motivation very aligned to ours in section II. They provide a tool to generate test models from a metamodel test specification, also in an ad-hoc language called TSMM (Test Specification Metamodel) Although both solutions provide checkers that allow the expert to verify properties on the metamodel and spot errors in his metamodel, the main drawback for both approaches

is the fact that a specification for model instances must be prepared beforehand, which is a manual process and thus it is not entirely automatic and can be expensive in large scale projects. Our automated approach to generate sets of test models might complement these works, by providing the user a qualified set of test data.

2) Grammar-Based Software Testing: Grammar-based software testing approaches work with the set of production rules of a grammar as an input. The goal is to achieve combinatorial coverage by using techniques of stochastic test-data generation. One such technique, proposed by Lämel et al. [20], deals with the problem by starting from full combinatorial coverage and subsequently managing control mechanisms to avoid combinatorial explosion, while ensuring coverage of the production rules. Another interesting application is proposed by Sirer et al. [21] with the purpose of testing the Java virtual machines, where a grammar for the language is used to generate test cases. The production rules of such grammar are accompanied by *certificates*, which act as the test oracle validating test cases.

VIII. Conclusions and Future Work

We have presented an approach to generate sets of test models in order to assist experts in the testing of metamodels. Such approach is bound to assist the construction of more *precise metamodels*, which capture a correct modeling space as intended by the domain expert. Our approach helps to spot errors in the metamodel definition, which improves this critical activity within Model-Driven Software Development activities.

In future work, we aim at building alternate implementation of our approach. As our problem deals with two distinct orthogonal objectives, another search-based technique to be tried out is Pareto Multi-Objective Optimization, among others.

In some cases, metamodels are accompanied with well-formedness rules in languages such as OCL, in order to express constraints of a complexity impossible to express in MOF. In future work, we aim at performing further validation taking into account such rules in the generation of sets of test models. This would allow us to give feedback to the expert not only about his metamodel, but also the defined well-formedness rules.

References

[1] *OMG Unified Modeling Language, v2.3*, Std., 2009.

[2] O. SPEM, "Software process engineering metamodel," *OMG Document Formal/02-11*, vol. 14, 2002.

[3] F. Fleurey, B. Baudry, P. Muller, and Y. Traon, "Qualifying input test data for model transformations," *Software and Systems Modeling*, vol. 8, no. 2, pp. 185–203, 2009.

[4] *OMG Meta Object Facility Core, v2.0*, Std., 2006.

[5] X. Thirioux, B. Combemale, X. Crégut, and P. Garoche, "A framework to formalise the mde foundations," *International Workshop on Towers of Models (TOWERS 2007)*, 2007.

[6] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis feasibility study," *Software Engineering Institute, Pittsburgh CMU/SEI-90-TR-21*, 1990.

[7] A. Arcuri, M. Iqbal, and L. Briand, "Black-box system testing of real-time embedded systems using random and search-based testing," *International Conference on Testing Software and Systems, ICTSS 2010*, pp. 95–110, 2010.

[8] F. Glover and G. Kochenberger, *Handbook of metaheuristics*. Springer, 2003.

[9] M. Harman, "The current state and future of search based software engineering," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 342–357.

[10] O. Räihä, "A survey on search-based software design," *Computer Science Review*, vol. 4, no. 4, pp. 203–249, 2010.

[11] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[12] S. Sen, B. Baudry, and J. Mottu, "Automatic model generation strategies for model transformation testing," *Theory and Practice of Model Transformations*, pp. 148–164, 2009.

[13] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "Uml2alloy: A challenging model transformation," *Model Driven Engineering Languages and Systems*, pp. 436–450, 2007.

[14] K. Dowsland, "Simulated annealing," in *Modern heuristic techniques for combinatorial problems*. John Wiley & Sons, Inc., 1993, pp. 20–69.

[15] F. Budinsky, E. Merks, and D. Steinberg, *Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.

[16] G. Perrouin, J. Klein, N. Guelfi, and J. Jézéquel, "Reconciling automation and flexibility in product derivation," in *Software Product Line Conference, 2008. SPLC'08. 12th International*. IEEE, pp. 339–348.

[17] T. Clark1, A. Evans, and S. Kent, "Engineering modelling languages: A precise meta-modelling approach," *Fundamental Approaches to Software Engineering*, pp. 242–260, 2002.

[18] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL models in USE by automatic snapshot generation," *Software and Systems Modeling*, vol. 4, no. 4, 2005.

[19] D. Sadilek and S. Weißleder, "Testing metamodels," in *Model Driven Architecture–Foundations and Applications*. Springer, 2008, pp. 294–309.

[20] R. Lämmel and W. Schulte, "Controllable combinatorial coverage in grammar-based testing," *Testing of Communicating Systems*, pp. 19–38, 2006.

[21] E. Sirer and B. Bershad, "Using production grammars in software testing," in *ACM SIGPLAN Notices*, vol. 35, no. 1. ACM, 1999, pp. 1–13.