Iterative Generation of Diverse Models for Testing Specifications of DSL Tools

No Author Given

No Institute Given

Abstract. The validation of modeling tools of custom domain-specific languages (DSLs) frequently relies upon an automatically generated set of models as a test suite. While many software testing approaches recommend that this test suite should be diverse, model diversity has not been studied systematically for graph models. In the paper, we propose diversity metrics for a single model as well as a set of models by exploiting the powerful abstraction technique of neighborhood shapes. Furthermore, we propose an iterative model generation technique to synthesize a diverse set of models where each model is taken from a different equivalence class as defined by neighborhood shapes. We evaluate our diversity metrics in the context of mutation testing for an industrial DSL and compare our model generation technique with a model generator based on Alloy which uses symmetry breaking predicates to enforce model diversity.

1 Introduction

Motivation. Domain-Specific Language (DSL) based modeling tools are gaining an increasing role in the software development processes. Advanced DSL frameworks such as Xtext, or Sirius built on top of model management frameworks such as Eclipse Modeling Framework (EMF) [38] significantly improve productivity by automating the production of rich editor features (e.g. syntax highlighting, auto-completion, etc.) to enhance modeling for domain experts.

Modelling environments may provide validation for the system under design from an early stage of development with efficient tool support for checking well-formedness (WF) constraints and design rules over large model instances of the DSL using tools like Eclipse OCL [27] or graph queries [42]. Model generation techniques [18,40,21,36] are able to automatically provide a range of solution candidates for allocation problems [21], model refactoring or context generation [24]. Finally, models can be are processed by query-based transformations or code generators to automatically synthesize source code or other artifacts.

The design of complex DSLs tools is a challenging task. As the complexity of DSL tools increases, special attention is needed to validate the modeling tools themselves (e.g. for tool qualification purposes) to ensure that WF constraints and the preconditions of model transformation and code generation functionality [5,36,34] are correctly implemented in the tool.

Problem Statement. There are many approaches aiming to address the testing of DSL tools (or transformations) [8,2,43] which necessitate *the automated* *synthesis of graph models* to serve as test inputs. Many best practices of testing (such as equivalence partitioning [28], mutation testing [20]) recommends the synthesis of *diverse* graph models where any pairs of models are structurally different from each other to achieve high coverage or a diverse solution space.

However, while software diversity is a widely studied [6], existing diversity metrics for graph models are much less elaborate [44]. Model comparison techniques [39] frequently rely upon the existence of node identifiers, which can easily lead to many isomorphic models. Moreover, checking graph isomorphism is computationally very costly. Therefore practical solutions tend to use approximate techniques to achieve certain diversity by random sampling [19], incremental generation [21,36], or using symmetry breaking predicates [40]. Unlike equivalence partitions which capture diversity of inputs in a customizable way for testing traditional software, a similar diversity concept is still missing for graph models.

Contribution. In this paper, we propose *diversity metrics* to characterize a single model and a set of models. For that purpose, we innovatively reuse neighborhood graph shapes [30], which provide a fine-grained typing for each object based on the structure (e.g. incoming and outgoing edges) of its neighborhood. Moreover, we propose an *iterative model generation technique* to automatically synthesize a diverse set of models for a DSL where each model is taken from a different equivalence class wrt. graph shapes as an equivalence relation.

We evaluate our diversity metrics and model generator in the context of mutation-based testing [25] of WF constraints in an industrial DSL tool. We evaluate and compare the *mutation score* and *our diversity metrics* of test suites obtained by (1) an Alloy based model generator (using symmetry breaking predicates to ensure diversity), (2) an iterative graph solver based generator using neighborhood shapes, and (3) from real models created by humans. Our finding is that a diverse set of models derived by along different neighborhood shapes is has better mutation score and also a better score wrt. our diversity metrics. Furthermore, based on a test suite with 4850 models, we found that high score in our diversity metrics would likely mean high mutation score which indicates that our metrics may be good predictors in practice for testing.

Added Value. Up to our best knowledge, our paper is one of the first studies on (software) model diversity. From a testing perspective, our diversity metrics provide a stronger characterization of a test suite of models than traditional metamodel coverage which is used in many research papers. Furthermore, model generators using on neighborhood graph shapes (that keep models only if they are surely non-isomorphic) provide increased diversity compared to symmetry breaking predicates (which exclude models if they are surely isomorphic).

2 Preliminaries

Core modeling concepts and testing challenges of DSL tools will be illustrated in the context of Yakindu Statecharts [46], which is an industrial DSL for developing reactive, event-driven systems using statecharts captured in a combined

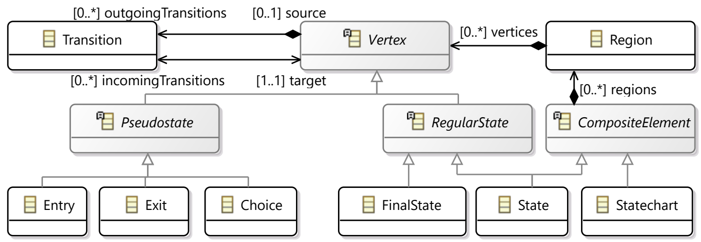


Fig. 1: Metamodel extract from Yakindu state machines

graphical and textual syntax. Yakindu simultaneously supports validation of well-formedness (WF) constraints and code generation from statechart models.

2.1 Metamodels and instance models

Metamodels define the main concepts, relations and attributes of a domain to specify the basic graph structure of models. A simplified metamodel for Yakindu state machines is illustrated in Figure 1 using the popular Eclipse Modeling Framework (EMF) [38] is used for domain modeling. A state machine consists of `Regions`, which in turn contain states (called `Vertexes`) and `Transitions`. An abstract state `Vertex` is further refined into `RegularStates` (like `State` or `FinalState`) and `PseudoStates` (like `Entry`, `Exit` or `Choice`).

Formally [34,35], a metamodel defines a vocabulary of type and relation symbols $\Sigma = \{\mathtt{C}_1, \ldots, \mathtt{C}_n, \mathtt{R}_1, \ldots, \mathtt{R}_m\}$ where a unary predicate symbol $\mathtt{C}_i$ is defined for each *EClass*, and a binary predicate symbol $\mathtt{R}_j$ is derived for each *EReference*. For space considerations, we omit the precise handling of attributes from this paper, which could be introduced accordingly.

An *instance model* can be represented as a logic structure $M = \langle Obj_M, \mathcal{I}_M \rangle$ where Obj_M is the finite set of objects (the size of the model is $|M| = |Obj_M|$), and $\mathcal{I}_M$ provides interpretation for all predicate symbols in Σ as follows:

- the interpretation of a unary predicate symbol $\mathtt{C}_i$ is defined in accordance with the types of the EMF model: $\mathcal{I}_M(\mathtt{C}_i) : Obj_M \to \{1, 0\}$
- the interpretation of a binary predicate symbol $\mathtt{R}_j$ is defined in accordance withe the links in the EMF model: $\mathcal{I}_M(\mathtt{R}_j) : Obj_M \times Obj_M \to \{1, 0\}$

A metamodel also specifies extra structural constraints. First, each object has a single direct type, i.e. for all object o there is a single non-abstract class $\mathtt{C}_D$ which may, in turn, have supertypes such as $\mathtt{C}_S(o) \Leftrightarrow [\mathtt{C}_S \textit{ supertype of } \mathtt{C}_D]$. Then, the *multiplicity* of structural features can be limited with upper and lower bounds: “lower..upper”. Then two parallel references of opposite direction can be defined as *inverses*: a pair of references `forw` and `back` are inverse of each other if $\mathtt{forw}(o, t) \Leftrightarrow \mathtt{back}(t, o)$. Finally, the EMF instance models are arranged into a strict *containment hierarchy*, which is a directed tree along relations marked in the metamodel as containment (see e.g. `regions` or `vertices` in the metamodel).

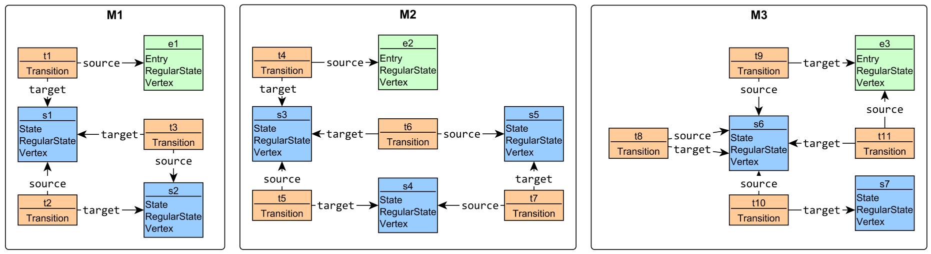


Fig. 2: Example instance models (as directed graphs)

$$
\begin{array}{rlrl}
[\![\texttt{C}(v)]\!]^M_Z & := \mathcal{I}_M(\texttt{C})(Z(v)) & [\![\varphi_1 \wedge \varphi_2]\!]^M_Z & := [\![\varphi_1]\!]^M_Z \wedge [\![\varphi_2]\!]^M_Z \\
[\![\texttt{R}(v_1, v_2)]\!]^M_Z & := \mathcal{I}_M(\texttt{R})(Z(v_1), Z(v_2)) & [\![\varphi_1 \vee \varphi_2]\!]^M_Z & := [\![\varphi_1]\!]^M_Z \vee [\![\varphi_2]\!]^M_Z \\
[\![v_1 = v_2]\!]^M_Z & := Z(v_1) = Z(v_2) & [\![\neg\varphi]\!]^M_Z & := \neg[\![\varphi]\!]^M_Z \\
[\![\forall v : \varphi]\!]^M_Z & := \bigwedge_{x \in Obj_M} [\![\varphi]\!]^M_{Z, v \mapsto x} & [\![\exists v : \varphi]\!]^M_Z & := \bigvee_{x \in Obj_M} [\![\varphi]\!]^M_{Z, v \mapsto x}
\end{array}
$$

Fig. 3: Inductive semantics of graph predicates

Example 1. Figure 2 shows graph representations of three instance models. In M_1 there are two `State`s ($s1$ and $s2$), which are connected to a loop via `Transition`s $t2$ and $t3$. The initial state is marked by a `Transition` $t1$ from an entry $e1$ to state $s1$. M_2 describes a similar statechart with three states in loop ($s3$, $s4$ and $s5$ connected via $t5$, $t6$ and $t7$). Finally, in M_3 there are two main differences: there is an incoming `Transition` $t11$ to an `Entry` state ($e3$), and there is a `State` $s7$ that does not have outgoing transition. While all these $M1$ and $M2$ are non-isomorphic, later we illustrate why they are not diverse.

2.2 Well-formedness Constraints as Logic Formulae

In many industrial modeling tools, WF constraints are captured either by OCL constraints [27] or graph patterns (GP) [22,42] where the latter captures structural conditions over an instance model as paths in a graph. To have a unified and precise handling of evaluating WF constraints, we use a tool-independent logic representation (which was influenced by [31,35,34]) that covers the key features of concrete graph pattern languages and a first-order fragment of OCL.

Syntax. A graph predicate is a first order logic predicate $\varphi(v_1, \dots v_n)$ over (object) variables which can be inductively constructed (see Figure 3) by using class and relation predicates $\texttt{C}(v)$ and $\texttt{R}(v_1, v_2)$, equality check $=$, standard first order logic connectives $\neg$, $\vee$, $\wedge$, and quantifiers $\exists$ and $\forall$.

Semantics. A graph predicate $\varphi(v_1, \dots, v_n)$ can be evaluated on model M along a variable binding $Z : \{v_1, \dots, v_n\} \to Obj_M$ from variables to objects in M. The truth value of φ can be evaluated over model M along the mapping Z (denoted by $[\![\varphi(v_1, \ldots, v_n)]\!]^M_Z$) in accordance with the semantic rules defined in Figure 3.

If there is a variable binding Z where the predicate φ is evaluated to 1 over M is often called a *pattern match*, formally $[\![\varphi]\!]^M_Z = 1$. Otherwise, if there are no bindings Z to satisfy a predicate, i.e. $[\![\varphi]\!]^M_Z = 0$ for all Z, then the predicate φ is evaluated to 0 over M. Graph query engines like [42,7] can retrieve (one or all) matches of a graph predicate over a model. When using graph patterns for validating WF constraints, a match of a pattern usually denotes a violation, thus the corresponding graph formula needs to capture the erroneous case.

2.3 Motivation: Mutation testing of DSL tools

A code generator would normally assume that the input models are well-formed, i.e. all WF constraints are validated prior to calling the code generator. However, there is no guarantee that the WF constraints actually checked by the DSL tool are exactly the same as the ones required by the code generator. For instance, if the validation forgets to check a subclause of a WF constraint, then runtime errors may occur during code generation. Moreover, the precondition of the transformation rule may also contain errors. For that purpose, WF constraints and model transformations of DSL tools can be systematically tested.

A popular approach for testing DSL tools is mutation testing [25,37] which aims to reveal missing or extra predicates by (1) deriving a set of mutants (e.g. WF constraints in our case) by applying a set of mutation operators. Then (2) the test suite is executed for both the original and the mutant programs, and (3) their output are compared. (4) A mutant is killed by a test if different output is produced for the two cases (i.e. different match set). (5) The mutation score of a test suite is calculated as the ratio of mutants killed by some tests wrt. the total number of mutants. A test suite with better mutation score is preferred [20].

Fault model and detection. As a fault model, we consider omission faults in WF constraints of DSL tools where some subconstraints are not actually checked. In our fault model, a WF constraint is given in a conjunctive normal form $\varphi_e = \varphi_1 \land \cdots \land \varphi_k$, all unbound variables are quantified existentially ($\exists$), and may refer to other predicates specified in the same form. Note that his format is equivalent to first order logic, and does not reduce the range of supported graph predicates. We assume that in a faulty predicate (a mutant) the developer may forget to check one of the predicates φ_i (Constraint Omission, CO), i.e. $[\varphi_1 \land \ldots \land \varphi_i \land \ldots \land \varphi_k]$ is rewritten to $[\varphi_1 \land \cdots \land \varphi_{i-1} \land \varphi_{i+1} \land \cdots \land \varphi_k]$, or may forgot a negation (Negation Omission), i.e. $[\varphi_1 \land \ldots \land (\neg\varphi_i) \land \ldots \land \varphi_k]$ is rewritten to $[\varphi_1 \land \ldots \land \varphi_i \land \ldots \land \varphi_k]$. Given an instance model M, we assume that both $[\![\varphi_e]\!]^M$ and $[\![\varphi_f]\!]^M$ can be evaluated separately by the DSL tool. Now a test model M detects a fault if there is a variable binding Z, where the two evaluations differ, i.e. $[\![\varphi_e]\!]^M_Z \neq [\![\varphi_f]\!]^M_Z$.

Example 2. Two WF constraints checked by the Yakindu environment can be captured by graph predicates as follows:

– φ : $incomingToEntry(E) := \exists T : \texttt{Entry}(E) \land \texttt{target}(T, E)$
– ϕ : $noOutgoingFromEntry(E) := \texttt{Entry}(E) \land \neg(\exists T : \texttt{source}(T, E))$

According to our fault model, we can derive two mutants for *incomingToEntry* as predicates $\varphi_{f_1} := \texttt{Entry}(E)$ and $\varphi_{f_2} := \exists t : \texttt{target}(T, E)$.

Constraints φ and ϕ are satisfied in model M_1 and M_2 as the corresponding graph predicates have no matches, thus $[\![\varphi]\!]_Z^{M_1} = 0$ and $[\![\phi]\!]_Z^{M_1} = 0$. As a test model, both M_1 and M_2 is able to detect the same omission fault both for φ_{f_1} as $[\![\varphi_{f_1}]\!]^{M_1} = 1$ (with $E \mapsto e1$ and $E \mapsto e2$) and similarly φ_{f_2} (with $s1$ and $s3$). However, M_3 is unable to kill mutant φ_{f_1} as (φ had a match $E \mapsto e3$ which remains in φ_{f_1}), but able to detect others.

3 Model Diversity Metrics for Testing DSL Tools

As a general best practice in testing, a good test suite should be diverse, but the interpretation of diversity may differ. For example, equivalence partitioning [28] partitions the input space of a program into equivalence classes based on observable output, and then select the different test cases of a test suite from different execution classes to achieve a diverse test suite. However, while software diversity has been studied extensively [6], model diversity is much less covered.

In existing approaches [11,12,43,33,9,8] for testing DSL and transformation tools, a test suite should provide full *metamodel coverage* [45], and it should also guarantee that any pairs of models in the test suite are non-isomorphic [19,40]. In [44], the diversity of a model M_i is defined as the number of (direct) types used from its MM, i.e. M_i is more diverse than M_j if more types of MM are used in M_i than in M_j. Furthermore, a model generator *Gen* deriving a set of models $\{M_i\}$ is diverse if there is a designated distance between each pairs of models M_i and M_j: $dist(M_i, M_j) > D$, but no concrete distance function is proposed.

Below, we propose diversity metrics for a single model, for pairs of models and for a set of models based on neighborhood shapes [30], a formal concept known from the state space exploration of graph transformation systems [29]. Our diversity metrics generalize both metamodel coverage and (graph) isomorphism tests, which are derived as two extremes of the proposed metric, and thus it defines a finer grained equivalence partitioning technique for graph models.

3.1 Neighborhood shapes of graphs

A neighborhood Nbh_i describes the local properties of an object in a graph model for a range of size $i \in \mathbb{N}$ [30]. The set of neighborhood descriptors are defined recursively with the set of class and reference symbols Σ:

– For range $i = 0$, Nbh_0 is a subset of class symbols: $Nbh_0 \subseteq 2^{\{\texttt{C}_1,\ldots,\texttt{C}_n\}}$
– A neighbor Ref_i for $i > 0$ is defined by a reference symbol and a neighborhood: $Ref_i \subseteq \{\texttt{R}_1, \ldots, \texttt{R}_m\} \times Nbh_{i-1}$.
– For a range $i > 0$ neighborhood Nbh_i is defined by a previous neighborhood and two sets of neighbor descriptors (for incoming and outgoing references separately): $Nbh_i \subseteq Nbh_{i-1} \times 2^{Ref_i} \times 2^{Ref_i}$.

Shaping function $nbh_i : Obj_M \rightarrow Nbh_i$ maps each object in a model M to a neighborhood with range i: (1) if $i = 0$, then $nbh_0(o) = \{\mathtt{C} | [\![\mathtt{C}(o)]\!]^M = 1\}$; (2) if $i > 0$, then $nbh_i(o) = \langle nbh_{i-1}(o), in, out\rangle$, where

$$in = \{\langle \mathtt{R}, n\rangle | \exists o' \in Obj_M : [\![\mathtt{R}(o', o)]\!]^M \wedge n = nbh_{i-1}(o')\}$$

$$out = \{\langle \mathtt{R}, n\rangle | \exists o' \in Obj_M : [\![\mathtt{R}(o, o')]\!]^M \wedge n = nbh_{i-1}(o')\}$$

A *(graph) shape* of a model M for range i (denoted as $S_i(M)$) is a set of neighborhood descriptors of the model: $S_i(M) = \{x | \exists o \in Obj_M : nbh_i(o) = x\}$. We will use the size of a shape $|S_i(M)|$ which is the number of shapes used in M. After calculating the neighborhood for each object, each neighborhood is represented as a node in the graph shape. Moreover, if there exist at least one link between objects in two different neighborhoods, the corresponding nodes in the shape will be connected by an edge. Informally, a node in the graph shape can be interpreted as a rich type graph where the original metamodel classes are split into multiple node types based on their potential neighborhood.

Example 3. We illustrate the concept of graph shapes for model M_1. For range 0, objects are mapped to class names as neighborhood descriptors:

- $nbh_0(e) = \{\mathtt{Entry}, \mathtt{PseudoState}, \mathtt{Vertex}\}$
- $nbh_0(t1) = nbh_0(t2) = nbh_0(t3) = \{\mathtt{Transition}\}$
- $nbh_0(s1) = nbh_0(s2) = \{\mathtt{State}, \mathtt{RegularState}, \mathtt{Vertex}\}$

For range 1, objects with different incoming or outgoing types are further split, e.g. the neighborhood of $t1$ is different from that of $t2$ and $t3$ as it is connected to an `Entry` along a `source` reference, while the `source` of $t2$ and $t3$ are `State`s.

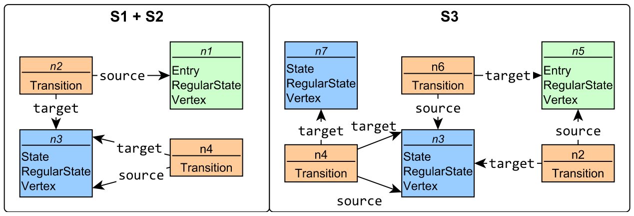
- $nbh_1(t1) = \langle\{\mathtt{Transition}\}, \emptyset, \{\langle\mathtt{source}, \{\mathtt{Entry}, \mathtt{PseudoState}, \mathtt{Vertex}\}\rangle, \langle\mathtt{target}, \{\mathtt{State}, \mathtt{RegularState}, \mathtt{Vertex}\}\rangle$
- $nbh_1(t2) = \langle\{\mathtt{Transition}\}, \emptyset, \{\langle\mathtt{source}, \{\mathtt{State}, \mathtt{RegularState}, \mathtt{Vertex}\}\rangle, \langle\mathtt{target}, \{\mathtt{State}, \mathtt{RegularState}, \mathtt{Vertex}\}\rangle = nbh_1(t3)$

For range 2, each object of M_1 would be mapped to a unique element. In Figure 4, the neighborhood shapes of models M_1, M_2, and M_3 for range 1, are represented in a visual notation adapted from [30,31] (without additional annotations e.g. multiplicities or predicates used for verification purposes). The trace of the concrete graph nodes to neighbourhood is illustrated on the right. For instance, $e1$ and $e2$ in $M1$ and M_2 `Entri`es are both mapped to the same neighbourhood $n1$, while $e3$ can be distinguished from them as it has incoming reference from a transition, thus creating a different neighbourhood $n5$.

Properties of graph shapes The theoretical foundations of graph shapes [30,31] prove several key semantic properties which are exploited in this paper:

P1 There are only a *finite number of graph shapes in a certain range*, and a smaller range reduces the number of graph shapes, i.e. $|S_i(M)| \leq |S_{i+1}(M)|$.

P2 $|S_i(M_j)| + |S_i(M_k)| \geq |S_i(M_j \cup M_k)| \geq |S_i(M_j)|$ and $|S_i(M_k)|$.



o	$nbh_1(o)$
$e1-2$	$n1$
$e3$	$n5$
$s1-6$	$n3$
$s7$	$n7$
$t1, 4, 11$	$n2$
$t2, 3, 5-8, 10$	$n4$
$t9$	$n6$

Fig. 4: Sample neighborhood shapes of M_1, M_2 and M_3

3.2 Metrics for model diversity

We define two metrics for model diversity based upon neighborhood shapes. *Internal diversity* captures the diversity of a single model, i.e. it can be evaluated individually for each and every generated model. As neighborhood shapes introduce extra subtypes for objects, this model diversity metric measures the number of neighborhood types used in the model with respect to the size of the model. *External diversity* captures the distance between pairs of models. Informally, this diversity distance between two models will be proportional to the number of different neighborhoods covered in one model but not the other, i.e. the number of model elements that fall into a different equivalence class.

Definition 1 (Internal model diversity). *For a range i of neighborhood shapes for model M, the internal diversity of M is the number of shapes wrt. the size of the model: $d_i^{int}(M) = |S_i(M)|/|M|$.*

The range of this internal diversity metric $d_i^{int}(M)$ is [0..1], and a model M with $d_1^{int}(M) = 1$ (and $|M| \geq |MM|$) *guarantees full metamodel coverage* [45], i.e. it surely contains all elements from a metamodel as types. As such, it is an appropriate diversity metric for a model in the sense of [44]. Furthermore, given a specific range i, the number of potential neighborhood shapes within that range is finite, but it grows superexponentially. Therefore, for a small range i, one can derive a model M_j with $d_i^{int}(M_j) = 1$, but for larger models M_k (with $|M_k| > |M_j|$) we will likely have $d_i^{int}(M_j) \geq d_i^{int}(M_k)$. However, due to the rapid growth of the number of shapes for increasing range i, for most practical cases, $d_i^{int}(M_j)$ will converge to 1 if M_j is sufficiently diverse.

Definition 2 (External model diversity). *Given a range i of neighborhood shapes, the external diversity of models M_j and M_k is the number of shapes contained exclusively in M_j or M_k but not in the other, formally, $d_i^{ext}(M_j, M_k) = |S_i(M_j) \oplus S_i(M_k)|$ where $\oplus$ denotes the symmetric difference of two sets.*

External model diversity allows to compare two models. One can show that this metric is a (pseudo)-distance in the mathematical sense [3], and thus, it can serve as a diversity metric for a model generator in accordance with [44].

Definition 3 (Pseudo-distance). *A function $d : \mathcal{M} \times \mathcal{M} \to \mathbb{R}$ is called a (pseudo-)distance, if it satisfies the following properties:*

- *d is non-negative: $d(M_j, M_k) \geq 0$*
- *d is symmetric $d(M_j, M_k) = d(M_k, M_j)$*
- *if M_j and M_k are isomorphic, then $d(M_j, M_k) = 0$*
- *triangle inequality: $d(M_j, M_l) \leq d(M_k, M_j) + d(M_j, M_l)$*

Corollary 1. *External model diversity $d_i^{ext}(M_j, M_k)$ is a (pseudo-)distance between models M_j and M_k for any i.*

During model generation, we will exclude a model M_k if $d_i^{ext}(M_j, M_k) = 0$ for a previously defined model M_j, but *it does not imply that they are isomorphic.* Thus our definition allows to avoid graph isomorphism checks between M_j and M_k which have high computation complexity. Note that external diversity is a dual of symmetry breaking predicates [40] used in the Alloy Analyzer where $d(M_j, M_k) = 0$ implies that M_j and M_k are isomorphic (and not vice versa).

Definition 4 (Coverage of model set). *Given a range i of neighborhood shapes and a set of models $MS = \{M_1, \ldots, M_k\}$, the coverage of this model set is defined as $cov_i\langle MS\rangle = |S_i(M_1) \cup \cdots \cup S_i(M_k)|$.*

The coverage of a model set is not normalised, but its value monotonously grows for any range i by adding new models. Thus it corresponds to our expectation that adding a new test case to a test suite should increase its coverage.

Example 4. Let us calculate the different diversity metrics for M_1, M_2 and M_3 of Figure 2. For range 1, they have the shapes illustrated in Figure 4. The internal diversity of those models are $d_1^{int}(M_1) = 4/6$, $d_1^{int}(M_2) = 4/8$ and $d_1^{int}(M_3) = 6/7$, thus M_3 is the most diverse model among them. As M_1 and M_2 has the same shape, the distance between them is $d_1^{ext}(M_1, M2) = 0$. The distance between M_1 and M_3 is $d_1^{ext}(M_1, M3) = 4$ as M_1 has 1 different neighbourhoods ($n1$), and M_3 has 3 ($n5$, $n6$ and $n7$). The set coverage of M_1, M_2 and M_3 is 7 altogether, as they have 7 different neighbourhoods ($n1$ to $n7$).

4 Iterative Generation of Diverse Models

Now we aim at generating a diverse set of models $MS = \{M_1, M_2, \ldots, M_k\}$ for a given metamodel MM (and potentially, a set of constraints WF). Our approach (see Figure 5) intentionally reuses several components as building blocks obtained from existing research results aiming to derive consistent graph models. First, model generation is an iterative process where previous solutions serve as further constraints [36]. Second, it repeatedly calls a back-end graph solver [1] to automatically derive consistent instance models which satisfy WF.

As a key conceptual novelty, we enforce the structural diversity of models during the generation process using neighborhood shapes at different stages. Most importantly, if the shape $S_i(M_k)$ of a new instance model M_n obtained

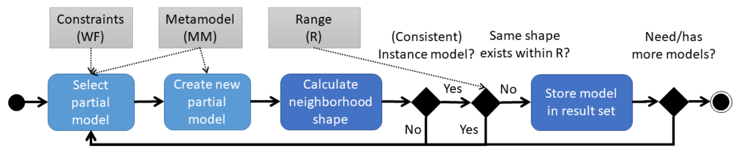


Fig. 5: Generation of diverse models

as a candidate solution is identical to the shape $S_i(M_j)$ for a previously derived model M_j for a predefined (input) neighborhood range i, the solution candidate is discarded, and iterative generation continues towards a new candidate.

Internally, our tool operates over partial models [32,35] where instance models are derived along a refinement calculus [44]. The shapes of intermediate (partial) models found during model generation are continuously being computed. As such, they may help guide the search process of model generation by giving preference to refine (partial) model candidates that likely result in a different graph shape. Furthermore, this extra bookkeeping also pays off once a model candidate is found since comparing two neighborhood shapes is fast (conceptually similar to lexicographical ordering). However, our concepts could be adapted to postprocess the output of other (black-box) model generator tools.

Example 5. As an illustration of the iterative generation of diverse models, let us imagine that model M_1 (in Figure 2) is retrieved first by a model generator. Shape $S_2(M_1)$ is then calculated (see Figure 4), and since there are no other models with the same shape, M_1 is stored as a solution. If the model generator retrieves M_2 as the next solution candidate, it turns out that $S_2(M_2) = S_2(M_1)$, thus M_2 is excluded. Next, if model M_3 is generated, it will be stored as a solution since $S_2(M_3) \neq S_2(M_2)$. Note that we intentionally omitted the internal search procedure of the model generator to focus on the use of neighborhood shapes.

Finally, it is worth highlighting that graph shapes are conceptually different from other approaches aiming to achieve diversity. Approaches relying upon object identifiers (like [39]) may classify two graphs which are isomorphic to be different. Sampling-based approaches [19] attempt to derive non-isomorphic models on a statistical basis, but there is no formal guarantee that two models are non-isomorphic. The Alloy Analyzer [40] uses *symmetry breaking predicates as sufficient conditions* of isomorphism (i.e. two models are surely isomorphic). *Graph shapes provide a necessary condition* for isomorphism i.e. if a two non-isomorphic models have identical shape, one of them is dissolved.

5 Evaluation

In this section, we provide an empirical evaluation of our diversity metrics and model generation technique to address the following research questions:

RQ1: How effective is our technique in creating diverse models for testing?
RQ2: How effective is our technique in creating diverse test suites?
RQ3: Is there correlation between diversity metrics and mutation score?

Target Domain. In order to answer those questions, we executed model generation campaigns on a DSL extracted from Yakindu Statecharts (as proposed in [36]). We used the partial metamodel describing the state hierarchy and transitions of statecharts (illustrated in Figure 1, containing 12 classes and 6 references). Additionally, we formalized 10 WF constraints regulating the transitions as graph predicates, based on the built-in validation of Yakindu.

For mutation testing, we used a constraint or negation omission operator (CO and NO) to inject an error to the original WF constraint in every possible way, which yielded 51 mutants from the original 10 constraints (but some mutants may never have matches). We checked both the original and mutated versions of the constraints for each instance model, and a model kills a mutant if there is a difference in the match set of the two constraints. The mutation score for a test suite (i.e. a set of models) is the total number of mutants killed that way.

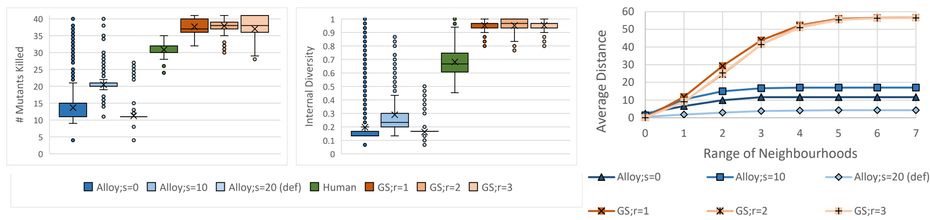
Compared approaches. Our test input models were taken from three different sources. First, we generated models with our iterative approach using a graph solver (**GS**) with different neighborhoods for ranges 1 to 3 (**r=1**,**r=2** and **r=3**).

Next, we generated models for the same DSL using **Alloy**[40], a well-known SAT-based relational model finder. For representing EMF metamodels we used traditional encoding techniques [10,34]. To enforce model diversity, Alloy was configured with three different setups for symmetry breaking predicates: **s=0**, **s=10** and **s=20** (default value). For greater values the tool produced the same set of models. We used the latest 4.2 build for Alloy with the default Sat4j [23] as back-end solver. All other configuration options were set to default.

Finally, we included 1250 manually created statechart models in our analysis (marked by **Human**). The models were created by students as solutions for similar (but not identical) statechart modeling homework assignments [44] representing real models which were *not* prepared for testing purposes.

Measurement setup. To address **RQ1-RQ3**, we created a two-step measurement setup. In **Step I.** a set of instance models is generated with all **GS** and **Alloy** configurations. Each tool in each configuration generated a sequence of 30 instance models produced by subsequent solver calls, and each sequence is repeated 20 times (so 1800 models are generated for both **GS** and **Alloy**). In case of **Alloy**, we prevented the deterministic run of the solver to enable statistical analysis. The model generators was to create metamodel-compliant instances compliant with the structural constraints of subsection 2.1 but ignoring the WF constraints. The target model size is set to 30 objects as Alloy did not scale with increasing size. The size of **Human** models ranges from 50 to 200 objects.

In **Step II.**, we evaluate and the mutation score for all the models (and for the entire sequence) by comparing results for the mutant and original predicates and record which mutant was killed by a model. We also calculate our diversity



(a) Mutation Score and Internal Diversity (b) External Diversity

Fig. 6: Mutation Scores and Diversity properties of models sets

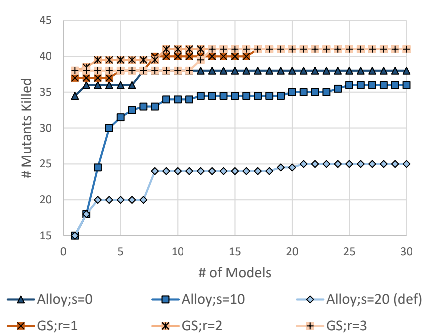
metrics for a neighborhood range where no more equivalence classes are produced by shapes (which turned out to be $r = 7$ in our case study). We calculated the internal diversity of each model, the external diversity (distance) between pairs of models in each model sequence, and the coverage of each model sequence.

RQ1: Measurement Results and Analysis Figure 6a shows the distribution of the number of mutants killed by at least one model from a model sequence (left box plot), and the distribution of internal diversity (right box plot). For killing mutants, **GS** was the best performer (regardless of the **r** range): most models found 36-41 mutants out of 51. On the other hand, **Alloy** performance varied based on the value of symmetry: for **s=0**, most models found 9-15 mutants (with a large number of positive outliers that found several errors). For **s=10**, the average is increased over 20, but the number of positive outliers simultaneously dropped. Finally, in default settings (**s=20**) **Alloy** generated similar models, and found only a low number of mutants. We also measured the efficiency of killing mutants by **Human**, which was between **GS** and **Alloy**. None of the the instance models could find more than 41 mutants, which suggests that those mutants cannot be detected at all by metamodel-compliant instances.

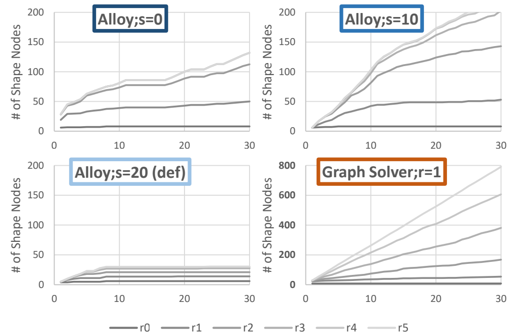
The right side of Figure 6a presents the internal diversity of models measured as shape nodes/graph nodes (for fixpoint range 7). The result are similar: the diversity was high with low variance in **GS** with slight differences between ranges. In case of **Alloy**, the diversity is similarly affected by the symmetry value: **s=0** produced low average diversity, but a high number of positive outliers. With **s=10**, the average diversity increased with decreasing number of positive outliers. And finally, with the default **s=20** value the average diversity was low. The internal diversity of **Human** models are between **GS** and **Alloy**.

Figure 6b illustrates the average distance between all model pairs generated in the same sequence (vertical axis) for range 7. The distribution of external diversity also shows similar characteristics as Figure 6a: **GS** provided high diversity for all ranges (56 out of the maximum 60), while the diversity between models generated by **Alloy** varied based on the symmetry value.

As a summary, our model generation technique consistently outperformed Alloy wrt. both the diversity metrics and mutation score for individual models.



(a) Mutation score for model sequence



(b) Model set coverage

Fig. 7: Mutation score and set coverage for model sequences

RQ2: Measurement Results and Analysis Figure 7a shows the number of killed mutants (vertical axis) by an increasing set of models (with 1 to 30 elements; horizontal axis) generated by **GS** or **Alloy**. The diagram shows the *median* of 20 generation runs to exclude the outliers. **GS** found a large amount of mutants in the first model, and the number of killed mutants (36-37) increased to 41 by the 17th model, which after no further mutants has been found. Again, our measurement showed little difference between ranges **r=1**, **2** and **3**. For **Alloy**, the result highly depends on the symmetry value: for **s=0** it found a large amount of mutants, but the value saturated early. Next, for **s=10**, the first model found significantly less mutants, but the number increased rapidly in the for the first 5 models, but altogether, less mutants were killed than for **s=0**. Finally, the default configuration (**s=20**) found the least number of mutants.

In Figure 7b, the average coverage of the model sets is calculated (vertical axis) for increasing model sets (horizontal axis). The neighborhood shapes are calculated for $r = 0$ to 5, which after no significant difference is shown. Again, configurations of symmetry breaking predicates resulted in different characteristics for **Alloy**. However, the number of shape nodes investigated by the test set was significantly higher in case of **GS**(791 vs. 200 equivalence classes) regardless of the range, and it was monotonously increasing by adding new models.

Altogether, both mutation score and equivalence class coverage of a model sequence was much better for our model generator approach compared to Alloy.

RQ3: Analysis of Results Figure 8 illustrates the correlation between mutation score (horizontal axis) and internal diversity (vertical axis) for all generated and human models in all configurations. Considering all models (1800 **Alloy**, 1800 **GS**, 1250 **Human**), mutation score and internal diversity shows a high correlation of 0.95 – while the correlation was low (0.12) for only **Human** models.

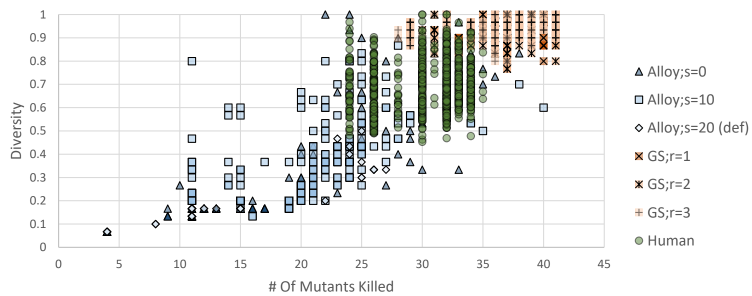


Fig. 8: Model diversity and mutation score correlation

Our initial investigation suggests that a high internal diversity will provide good mutation score, thus our metrics can potentially be good predictors in a testing context, but we cannot generalize to full statistical correlation.

Threats to Validity and Limitations. We evaluated more than 4850 test inputs in our measurement, but all models were taken from a single domain of Yakindu statecharts with a dedicated set of WF constraints. However, our model generation approach did not used any special property of the metamodel or the WF constraints, thus we believe that similar results would be obtained for other domains. For mutation operations, we checked only omission of predicates, as extra constraints could easily yield infeasible predicates due to inconsistency with the metamodel, thus further reducing the number of mutants that can be killed.

Finally, although we detected a strong correlation between diversity and mutation score with our test cases, this result cannot be generalized to statistical causality, because the generated models were not random samples taken from the universe of models. Thus additional investigations are needed to justify this correlation, and we only state that if a model is generated by either **GS** or **Alloy**, a higher diversity means a higher mutation score with high probability.

6 Related Work

Diverse models play a key role in testing model transformations and code generators. Mutation-based approaches [25,13,2] take existing models and make random changes on them by applying mutation rules. A similar random model generator is used for experimentation purposes in [4]. Other automated techniques [9,14] generate models that only conform to the metamodel. While these techniques scale well for larger models, there is no guarantee whether the mutated models satisfy WF constraints.

There is a wide set of model generation techniques which provide certain promises for test effectiveness. White-box approaches [2,8,16,17,33,34] rely on the implementation of the transformation and dominantly use back-end logic solvers, which lack scalability when deriving graph models.

Scalability and diversity of solver-based techniques can be improved by iteratively calling the underlying solver [21,36]. In each step a partial model is extended with additional elements as a result of a solver call. Higher diversity is achieved by avoiding the same partial solutions. As a downside, generation steps need to be specified manually, and higher diversity can be achieved only if the models are decomposable into separate well-defined partitions.

Black-box approaches [15,26,10,17] can only exploit the specification of the language or the transformation, so they frequently rely upon contracts or model fragments. As a common theme, these techniques may generate a set of simple models, and while certain diversity can be achieved by using symmetry-breaking predicates, they fail to scale for larger sizes. In fact, the effective diversity of models is also questionable since corresponding safety standards prescribe much stricter test coverage criteria for software certification and tool qualification than those currently offered by existing model transformation testing approaches.

Based on the logic-based Formula solver, the approach of [19] applies stochastic random sampling of output to achieve a diverse set of generated models by taking exactly one element from each equivalence class defined by graph isomorphism, which can be too restrictive for coverage purposes. Stochastic simulation is proposed for graph transformation systems in [41], where rule application is stochastic (and not the properties of models), but fulfillment of WF constraints can only be assured by a carefully constructed rule set.

7 Conclusion and Future Work

We proposed novel diversity metrics for models based on neighbourhood shapes [30], which are true generalizations of metamodel coverage and graph isomorphism used in many research papers. Moreover, we presented a model generation technique that to derive structurally diverse models by (i) calculating the shape of the previous solutions, and (ii) feeding back to an existing generator to avoid similar instances thus ensuring high diversity between the models.

We evaluated our approach in a mutation testing scenario for Yakindu Statecharts, an industrial DSL tool. We compared the effectiveness (mutation score) and the diversity metrics of different test suites derived by our approach and an Alloy-based model generator. Our approach consistently outperformed the Alloy-based generator for both a single model and the entire test suite. Moreover, we found high (internal) diversity values normally result in high mutation score, thus highlighting the practical value of the proposed diversity metrics.

Conceptually, our approach can be adapted to an Alloy-based model generator by adding formulae obtained from previous shapes to the input specification. However, our initial investigations revealed that such an approach does not scale well with increasing model size. While Alloy has been used as a model generator for numerous testing scenarios of DSL tools and model transformations [8,10,36,37,43], our measurements strongly indicate that it is not a justified choice as (1) Alloy is very sensitive to configurations of symmetry breaking predicates and (2) the diversity and mutation score of generated models is problematic.

References

1. Anonymous for double-blind version.
2. V. Aranega, J.-M. Mottu, A. Etien, T. Degueule, B. Baudry, and J.-L. Dekeyser. Towards an automation of the mutation analysis dedicated to model transformation. *Softw. Test., Verif. Reliab.*, 25(5-7):653–683, 2015.
3. A. Arkhangel'Skii and V. Fedorchuk. *General topology I: basic concepts and constructions dimension theory*, volume 17. Springer Science & Business Media, 2012.
4. E. Batot and H. Sahraoui. A generic framework for model-set selection for the unification of testing and learning MDE tasks. In *MODELS*, pages 374–384, 2016.
5. B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. Model transformation testing challenges. In *Integration of Model Driven Development and Model Driven Testing*, 2006.
6. B. Baudry, M. Monperrus, C. Mony, F. Chauvel, F. Fleurey, and S. Clarke. Diversify: Ecology-inspired software evolution for diversity emergence. In *Software Maintenance, Reengineering and Reverse Engineeringn*, pages 395–398, 2014.
7. E. Biermann, K. Ehrig, C. Ermel, C. Köhler, and G. Taentzer. The EMF model transformation framework. In *AGTIVE*, pages 566–567, 2007.
8. B. Bordbar and K. Anastasakis. Uml2alloy: A tool for lightweight modelling of discrete event systems. In *IADIS AC*, pages 209–216, 2005.
9. E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In *17th International Symposium on Software Reliability Engineering*, pages 85–94, 2006.
10. F. Büttner, M. Egea, J. Cabot, and M. Gogolla. Verification of ATL transformations using transformation models and model finders. In *14th International Conf. on Formal Engineering Methods*, pages 198–213, 2012.
11. J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE*, pages 547–548, 2007.
12. J. Cabot, R. Clariso, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *ICSTW*, pages 73–80, 2008.
13. A. Darabos, A. Pataricza, and D. Varró. Towards testing the implementation of graph transformations. In *GTVMT*, ENTCS. Elsevier, 2006.
14. K. Ehrig, J. M. Küster, and G. Taentzer. Generating instance models from meta models. *Softw. Syst. Model*, 8(4):479–500, 2009.
15. F. Fleurey, B. Baudry, P.-A. Muller, and Y. Le Traon. Towards dependable model transformations: Qualifying input test data. *SoSyM*, 2007.
16. C. A. Gonzalez and J. Cabot. Test data generation for model transformations combining partition and constraint analysis. In *ICMT*, pages 25–41, 2014.
17. E. Guerra and M. Soeken. Specification-driven model transformation testing. *Softw. Syst. Model.*, 14(2):623–644, may 2015.
18. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
19. E. K. Jackson, G. Simko, and J. Sztipanovits. Diversely enumerating system-level architectures. In *Int. Conf. on Embedded Software*, page 11, 2013.
20. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.
21. E. Kang, E. Jackson, and W. Schulte. An approach for effective design space exploration. In *Foundations of Computer Software.*, pages 33–54. 2011.
22. D. S. Kolovos, R. F. Paige, and F. A. C. Polack. On the evolution of OCL for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis*, pages 204–218. 2009.

23. D. Le Berre and A. Parrain. The Sat4j library. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.
24. Z. Micskei, Z. Szatmári, J. Oláh, and I. Majzik. A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In *KES-AMSTA*, volume 7327 of *LNCS*, pages 504–513. Springer, 2012.
25. J. Mottu, B. Baudry, and Y. L. Traon. Mutation analysis testing for model transformations. In *ECMDA-FA*, volume 4066 of *LNCS*, pages 376–390. Springer, 2006.
26. J.-M. Mottu, S. S. Simula, J. Cadavid, and B. Baudry. Discovering model transformation pre-conditions using automatically generated test models. In *ISSRE*, pages 88–99. IEEE, nov 2015.
27. The Object Management Group. *Object Constraint Language, v2.0*, May 2006.
28. S. C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Software Metrics Symposium*, pages 64–73, 1997.
29. A. Rensink. Isomorphism checking in GROOVE. *ECEASST*, 1, 2006.
30. A. Rensink and D. Distefano. Abstract graph transformation. *Electronic Notes in Theoretical Computer Science*, 157(1):39–59, 2006.
31. T. W. Reps, M. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In *International Conference on Computer Aided Verification*, pages 15–30, 2004.
32. R. Salay, M. Famelis, and M. Chechik. Language independent refinement using partial modeling. In *FASE*, volume 7212 of *LNCS*, pages 224–239. 2012.
33. J. Schonbock, G. Kappel, M. Wimmer, A. Kusel, W. Retschitzegger, and W. Schwinger. TETRABox - a generic white-box testing framework for model transformations. In *APSEC*, pages 75–82. IEEE, dec 2013.
34. O. Semeráth, A. Barta, A. Horváth, Z. Szatmári, and D. Varró. Formal validation of domain-specific languages with derived features and well-formedness constraints. *Software and Systems Modeling*, pages 1–36, 2015.
35. O. Semeráth and D. Varró. Graph constraint evaluation over partial models by constraint rewriting. In *ICMT*, pages 138–154, 2017.
36. O. Semeráth, A. Vörös, and D. Varró. Iterative and incremental model generation by logic solvers. In *FASE*, pages 87–103, 2016.
37. S. Sen, B. Baudry, and J. Mottu. Automatic model generation strategies for model transformation testing. In *ICMT*, pages 148–164, 2009.
38. The Eclipse Project. *Eclipse Modeling Framework*. `//www.eclipse.org/emf`.
39. The Eclipse Project. *EMF DiffMerge*. `wiki.eclipse.org/EMF_DiffMerge`.
40. E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. 2007.
41. P. Torrini, R. Heckel, and I. Ráth. Stochastic simulation of graph transformation systems. In *FASE*, volume 6013 of *LNCS*, pages 154–157. Springer, 2010.
42. Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98, 02/2015 2015.
43. A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann. Formal specification and testing of model transformations. In *SFM*, pages 399–437, 2012.
44. D. Varró, O. Semeráth, G. Szárnyas, and Ákos Horváth. Towards the automated generation of consistent, diverse, scalable and realistic graph models. In *Festschrift in Memory of Hartmut Ehrig*. Springer, 2017.
45. J. Wang, S.-K. Kim, and D. Carrington. Verifying metamodel coverage of model transformations. In *Software Engineering Conference*, pages 10–pp, 2006.
46. Yakindu Statechart Tools. *Yakindu*. `http://statecharts.org/`.