International Journal on Software Tools for Technology Transfer (2020) 22:57–78
https://doi.org/10.1007/s10009-019-00530-6

FASE 2018

Diversity of graph models and graph generators in mutation testing

Oszkár Semeráth[1,2] · Rebeka Farkas[1,2] · Gábor Bergmann[1,2] · Dániel Varró[1,2,3]

Published online: 11 September 2019

Abstract
When custom modeling tools are used for designing complex safety-critical systems (e.g., critical cyber-physical systems), the tools themselves need to be validated by systematic testing to prevent tool-specific bugs reaching the system. Testing of such modeling tools relies upon an automatically generated set of models as a test suite. While many software testing practices recommend that this test suite should be diverse, model diversity has not been studied systematically for graph models. In the paper, we propose different diversity metrics for models by generalizing and exploiting neighborhood and predicate shapes as abstraction. We evaluate such shape-based diversity metrics using various distance functions in the context of mutation testing of graph constraints and access policies for two separate industrial DSLs. Furthermore, we evaluate the quality (i.e., bug detection capability) of different (random and consistent) model generation techniques for mutation testing purposes.

Keywords Graph diversity metrics · Model diversity · Model generators · Mutation testing · Shape analysis

1 Introduction

Motivation. Several modeling tools in the industrial practice of model-based systems engineering (such as Capella, Artop, or Papyrus) are built upon graph-based model representations. Such modeling tool 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 domain-specific languages (DSL) using tools like Eclipse OCL [39] or graph queries [58]. They may provide support for complex model transformation or code generation steps to automatically derive or continuously maintain various design artifacts.

When such modeling tools are used for engineering complex, safety-critical systems (like safety-critical cyber-physical systems), they need to comply with related safety standards (like DO-178C) where tool qualification frequently prescribes that software tools used for engineering such systems also need to be tested systematically. Such testing requires a test suite that is of sufficiently high *quality*, i.e., it is capable of revealing a large fraction of the anticipated defects in the system, in an efficient manner. However, a test suite of a modeling tool is fundamentally different from that of a traditional software system since a test input needs to be a complex, graph-based model. Moreover, for advanced modeling environments, complex instance models are scarce due to the protection of intellectual property.

Various techniques have been proposed [29,33,48,51,56] for *the automated synthesis of graph models* to serve as test inputs for testing of model transformations, [3,11,59], for solving the allocation problems [33], for model refactoring or context generation [36] in CPS. Often, these model generators have to generate *consistent* models which sat-

This paper was partially supported by NSERC RGPIN-04573-16 Project and the 3rd author was partially supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences as well as the NKP-18-4 New National Excellence Program of the Ministry of Human Capacities.

✉ Oszkár Semeráth
semerath@mit.bme.hu

Rebeka Farkas
farkas@mit.bme.hu

Gábor Bergmann
bergmann@mit.bme.hu

Dániel Varró
varro@mit.bme.hu

[1] MTA-BME Lendület Cyber-Physical Systems Research Group, Budapest, Hungary

[2] Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary

[3] Department of Electrical and Computer Engineering, McGill University, Montreal, Canada

isfy well-formedness (WF) constraints, otherwise the models may not be processed by the modeling tool under test.

Many best practices of software testing (such as equivalence partitioning [41], mutation testing [32]) recommend the synthesis of *diverse* test inputs where each test case elicits structurally different behavior from the *artifact under test (AUT)* in order to achieve high coverage or a diverse solution space. In fact, mutation testing is frequently used [3,37,46] in model-driven engineering (MDE).

Problem Statement. While diversity is widely studied [7] for traditional software, existing diversity metrics for software models are much less elaborated [61]. Model comparison techniques [54] frequently rely upon the existence of node identifiers, which can easily lead to many isomorphic models. Since checking graph isomorphism is computationally costly, practical solutions tend to use approximation techniques to achieve certain diversity by random sampling [31], incremental generation [33,51], using symmetry-breaking predicates [56], or – in case of a white-box testing – by generating models satisfying (or violating) relevant graph predicates [51] (possibly derived from OCL expressions [22,26]).

In the current paper, we wish to investigate three major questions:

- Q1 *How to characterize diversity in case of graph models?* In other terms, we aim to study the question what makes a single model or a set of models diverse.
- Q2 *How representative or useful are diversity metrics for deriving an effective test suite?* In other terms, if we derive a set of models which are diverse wrt. some metrics, then ideally, this set of models should serve as an effective test suite for functional testing.
- Q3 *How effective are existing model generators in deriving models as a test suite for functional testing?* In other terms, to what an extent can existing generators help in deriving an effective test suite.

To measure the effectiveness of a test suite (or a single test case), we are using a mutation testing setup. First, we specify potential fault models for components of a modeling tool, then we derive a set of mutant artifacts by injecting possible faults to the modeling tool. A fault can be *killed* by an instance model if a mutant artifact handles the model differently from what we expected (which is decided by a *test oracle*). Therefore, the effectiveness of test cases, test suites and model generators can be measured by the number of mutant cases it can detect (so-called *mutation score*). In general, the goal is to create a test suite that is able to kill the most faults in mutation testing setup.

Contribution. In this paper (which extends our initial work in [50]), we propose and evaluate various *diversity metrics* to characterize a single model and a set of models. For that purpose, we *innovatively reuse neighborhood graph shapes* [43] and *graph predicate abstraction techniques* [22,26,44] which refine the type of each graph object based on the structure of its neighborhood (e.g., incoming and outgoing edges) and the truth values of predicates (e.g., a node satisfies a predicate or not). The *internal diversity* of a single model will be characterized by the number of shape nodes covered by the model, which generalizes traditional metamodel coverage (used in many research papers). Furthermore, we *adapt several distance metrics* (e.g., symmetric distance, cosine distance) to capture the diversity between pairs of graph models, and generalize shape-based *external diversity* metrics characterizing the whole test suite.

We also conduct experiments with a large number of instance models to investigate whether such metrics can successfully predict (or improve, via *test case prioritization* as in [9]) the quality of a test or test suite. Experiments are conducted by mutation-based testing [37], with *mutation score* as the proxy for test quality. Our experimental evaluation spans two case studies taken from different application domains:

- **ST** Testing a statechart-based DSL tool used for behavioral modeling (state machine models) as in [50]. Here, we investigated how diverse input models can test the correctness of the language specification of the the modeling tool.
- **WT** Testing access control policies for collaborative modeling (in the context of structural models of wind turbine control systems [23]). Here, we investigated how diverse input models can test the correctness of the access settings of a collaborative modeling platform.

As such, our paper is one of the first extensive studies on (software) model diversity.

Test inputs to be compared stem from four different sources: (1) an Alloy-based consistent model generator [56] (using symmetry-breaking predicates to ensure diversity), (2) a consistent model generator based on a graph solver [48] that uses neighborhood shapes, (3) a random model generator [45] without consistency guarantees, and finally (4) real models created by humans.

Based on a test suite with 23,439 models, we found *high correlation between mutation score and our diversity metrics*, which indicates that our metrics are likely to be good predictors for mutation testing purposes in MDE. Furthermore, model generators using neighborhood graph shapes (that keep models only if they are surely non-isomorphic) provide significantly better diversity (and mutation score) compared to symmetry-breaking predicates (which exclude

models if they are surely isomorphic). Since Alloy is following this second path, this empirical finding likely invalidates a significant amount of past research results (e.g. [3,11,24,25,46,47]) where the Alloy Analyzer was used for generate models, as those test inputs might be well formed, but not necessarily efficient (with respect to mutation score). Finally, with random restarts, the graph solver [48] derived more diverse graph models than a random EMF generator [45] and significantly outperformed Alloy-based model generators.

This paper extends previous work [50] by incorporating novel distance metrics, a new complex case study, and a substantially extended experimental evaluation (with new research questions, model generators and a new case study). Finally, as a by-product, we also provide a reusable mutation generator (following ideas in [10,35]) for a model-based access control policy language [18].

2 Preliminaries

Core modeling concepts and testing challenges of DSL tools will be illustrated in the context of Yakindu Statecharts [63], which is an industrial DSL for developing reactive, event-driven systems, and supports validation and code generation.

2.1 Case studies

The two cases aim at testing different kinds of DSL artifacts: WF constraints and model access policies, respectively. We measured the quality of the generated test suites in the context of mutation testing. First, we specified a fault model for the artifacts of both case studies, and we derived mutant modeling artifacts by injecting errors accordingly: we derived a set of mutant WF constraints for **ST**, and a set of mutated access control policies (for **WT**). Then, we applied both the original and mutant artifacts on both single test input models and complete test suites, and evaluated whether the test input is able to detect the fault injected into the mutant. A test input model detects the fault if it differentiates the mutant from the original, i.e., the two artifacts yield different outcomes for the model (respectively, resulting in different well-formedness violations, or different levels of access). The number of detected mutant artifacts is measured by a mutation score, which is considered a proxy for test suite quality.

Example 1 State machines describe state-based behaviors of systems by depicting the states and the possible transitions in-between. Yakindu Statecharts [63] encapsulate state machines in so-called *regions*. There are states with special roles (e.g., an *entry state* is the first state that becomes active when entering a region), collectively referred to as *pseudostates*. Transitions have a source and a target state that may be of any kind. Some simple state machines are depicted in abstract syntax (as labelled graphs) in Fig. 1.

A simplified metamodel for Yakindu state machines is illustrated in Fig. 2 using the popular Eclipse Modeling Framework (EMF) [53] used for domain modeling. A state machine consists of `Regions`, which in turn contain states (called `Vertexes`) and `Transitions`. The abstract state `Vertex` is further refined into `RegularStates` (like `State` or `FinalState`) and `PseudoStates` (like `Entry`, `Exit` or `Choice`).

Example 2 As a second case study of real-world industrial significance, let us consider a domain-specific modeling language [23] developed by IKERLAN for the model-driven development of control systems for wind turbines (WT). The core WT control domain metamodel is depicted in Fig. 3 (for now, ignore the three differently colored types on the left). The control system `WT` is composed of different `ControlUnits` as building blocks (each realizing a separate control algorithm), organized in a containment hierarchy of `Subsystems` (and `MainSubsystems`). Such control units (and, by virtue of containment, the subsystems themselves as well) consume `inputs` and produce `outputs`; such signals, from the point of view of the entire system, may be classified as `SystemInputs`, `SystemOutputs` or internal variables (`SystemVariables`). Control units are further characterized by a number of additional features including attributes such as `cycle` and cross-references to elements such as `SystemTimers`.

Access control requirements (detailed in the online appendix [1]) necessitate the storage of additional meta-information: There is a concept of *ownership*, where certain classes of specialists are assigned responsibility over of a `Subsystem` (and everything contained therein); while certain subsystems may express *protected* intellectual property (IP) and must be hidden from such specialist contractors. We have opted to extend the original metamodel to be able to seamlessly express this information in our experiments; hence, the new root container object `AuthorizedSystem`, the class `Ownership` and their various features were introduced. Since IKERLAN has opted not to disclose the list of their actual proprietary control unit types, we have combined all of them into a single class `ControlUnit` as a further deviation from the original. The reader is encouraged to compare Fig. 3 with the original metamodel diagram in [23].

[1] https://github.com/FTSRG/publication-pages/wiki/Diversity-STTT-2019

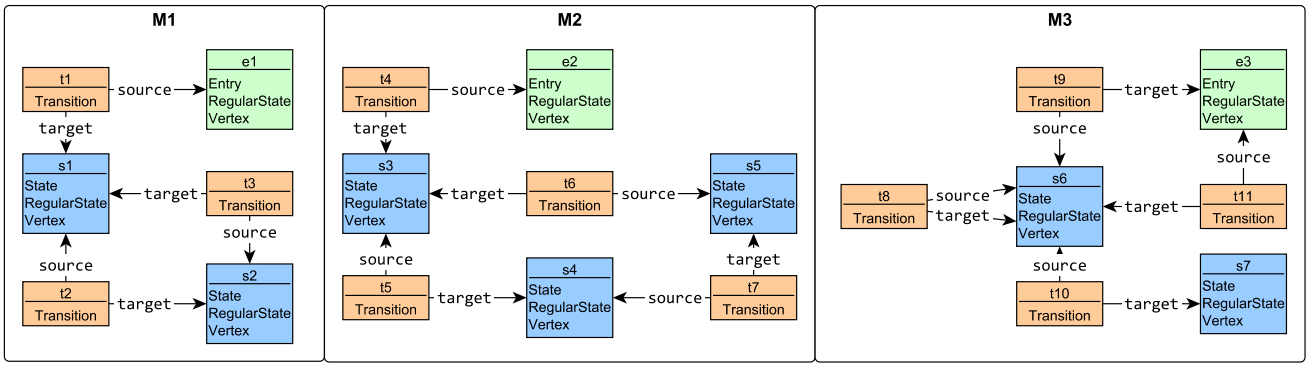


Fig. 1 Example instance models (as directed graphs)

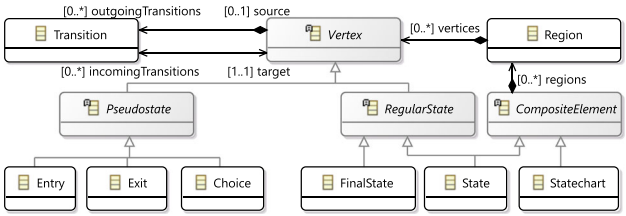


Fig. 2 Metamodel extract from Yakindu state machines

2.2 Metamodels and instance models

In this paper, we use EMF as a metamodeling technique which is widely used in the modeling community. Nevertheless, the presented techniques can either be analogously adapted to other metamodeling approaches, or applied directly after creating a representation of the model in EMF (like in case of UML model with Papyrus[55]). Formally [47,49], an (EMF) metamodel defines a vocabulary $\langle \Sigma, \alpha \rangle$ where $\Sigma = \{\mathrm{C}_1, \ldots, \mathrm{C}_n, \mathrm{R}_1, \ldots, \mathrm{R}_m, \mathrm{P}_1, \ldots, \mathrm{P}_l\}$ is a set of type, relation, and predicate symbols where a unary predicate symbol C_i is defined for each *EClass* and *EDataType* (like *EString*, *EInteger*, or *EEnum*), a binary predicate symbol R_j is derived for each *EReference* and *EAttribute* and additional graph predicates $\mathrm{P}_1, \ldots, \mathrm{P}_l$ are also provided. The additional predicates are a generalization of class and reference symbols where the number of parameters can be arbitrary. The function $\alpha : \Sigma \to \mathbb{N}$ assigns arities to predicates, such that $\alpha(\mathrm{C}_i) = 1$ and $\alpha(\mathrm{R}_j) = 2$ for all i, j.

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 C_i is defined in accordance with the types of the EMF model: $\mathcal{I}_M(\mathrm{C}_i) : Obj_M \to \{1, 0\}$. An object $o \in Obj_M$ is an instance of (more precisely, conforms to) a class C_i in a model M if $\mathcal{I}_M(\mathrm{C}_i)(o) = 1$. It is possible for an object to conform to multiple types, e.g., in case of inheritance or abstract classes. In EMF, it is required that each object conforms to at least one non-abstract type, and all of its supertypes. However, by stating that all supertypes and

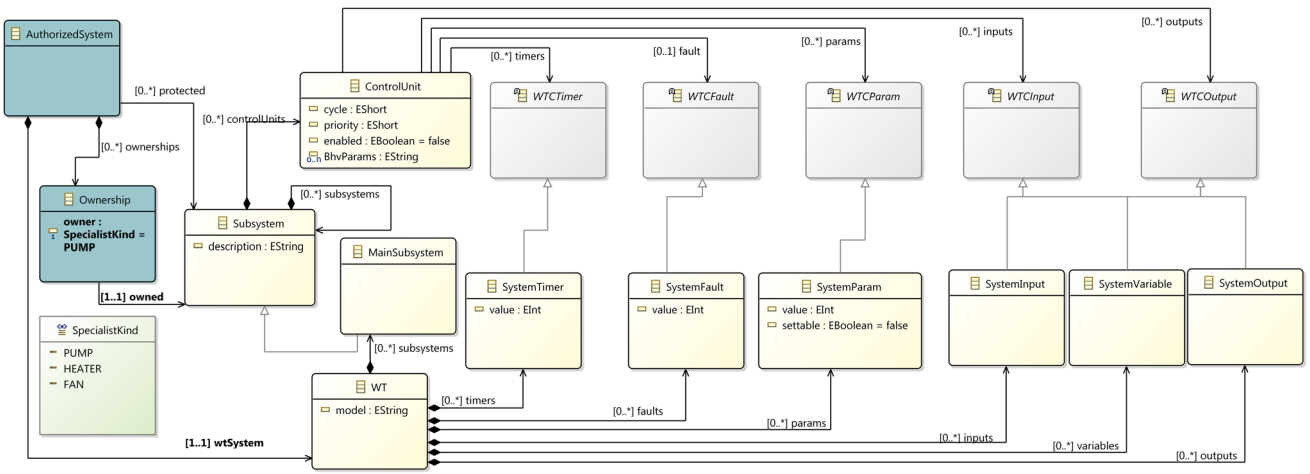


Fig. 3 Wind turbine DSL metamodel, adapted from [23]

subtypes are listed, it can also be applied to more general type systems.
- The interpretation of a binary predicate symbol $\mathtt{R}_j$ is defined in accordance with the links in the EMF model: $\mathcal{I}_M(\mathtt{R}_j) : Obj_M \times Obj_M \to \{1, 0\}$. There is a reference $\mathtt{R}_j$ between $o_1, o_2 \in Obj_M$ in model M if $\mathcal{I}_M(\mathtt{R}_j)(o_1, o_2) = 1$.
- The interpretation of a predicate symbol $\mathtt{P}_k$ is discussed in Sect. 2.3: $\mathcal{I}_M(\mathtt{P}_k) : Obj_M^{\alpha(\mathtt{P}_k)} \to \{1, 0\}$. There is a *match* on objects $o_1, o_2, \ldots, o_{\alpha(\mathtt{P}_k)} \in Obj_M$ if $\mathcal{I}_M(\mathtt{P}_k)(o_1, o_2, \ldots, o_{\alpha(\mathtt{P}_k)}) = 1$.

However, not all such logic structures constitute valid instance models. A metamodel also specifies extra structural constraints that need to be satisfied in each valid instance model [47]. Such structural constraints include the type conformance of nodes and edges as well as type hierarchy (essentially type conformance of the object graph), multiplicities, acyclic containment structure, etc., in addition to custom WF constraints.

Example 3 Figure 1 shows graph representations of three (partial) state machine instance models. For the sake of clarity, `Regions` and inverse (opposite) relations `incoming` `Transitions` and `outgoingTransitions` are excluded from the diagram. In M_1 there are two `States` ($s1$ and $s2$), which are connected to a loop via `Transitions` $t2$ 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. Additional constraints expressed in the metamodel, such as each transition having at most one source and one target, are also satisfied by the above graphs; therefore, they can be considered valid instance models. While all these instance models are non-isomorphic, later we illustrate why they are not diverse.

Some modeling technologies and metamodeling approaches use slightly different definitions for these concepts, but this makes little difference to the substance of our paper. For instance, in UML, instances of an Association are entities in their own right; they have attribute values and multiple such links can share the same source and target. In our approach, graph edges are represented as binary predicates instead. However, we can make this restriction without loss of generality: Instances of UML Associations can be adequately represented in our framework as objects (graph nodes), with distinguished outgoing edges representing the association ends. Similarly, definitions in the sequel apply to such alternate modeling formalisms as well.

$$
\begin{aligned}
[\![\mathtt{C}(v)]\!]_Z^M &:= \mathcal{I}_M(\mathtt{C})(Z(v)) \\
[\![\mathtt{R}(v_1, v_2)]\!]_Z^M &:= \mathcal{I}_M(\mathtt{R})(Z(v_1), Z(v_2)) \\
[\![v_1 = v_2]\!]_Z^M &:= Z(v_1) = Z(v_2) \\
[\![\varphi_1 \wedge \varphi_2]\!]_Z^M &:= [\![\varphi_1]\!]_Z^M \wedge [\![\varphi_2]\!]_Z^M \\
[\![\varphi_1 \vee \varphi_2]\!]_Z^M &:= [\![\varphi_1]\!]_Z^M \vee [\![\varphi_2]\!]_Z^M \\
[\![\neg\varphi]\!]_Z^M &:= \neg[\![\varphi]\!]_Z^M \\
[\![\forall v : \varphi]\!]_Z^M &:= \bigwedge_{x \in Obj_M} [\![\varphi]\!]_{Z, v \mapsto x}^M \\
[\![\exists v : \varphi]\!]_Z^M &:= \bigvee_{x \in Obj_M} [\![\varphi]\!]_{Z, v \mapsto x}^M
\end{aligned}
$$

Fig. 4 Inductive semantics of graph predicates

2.3 Graph predicates

In many industrial modeling tools, WF constraints are captured either by OCL constraints [39] or graph patterns (GP) [58] where the latter captures errors as 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 [44,47,49]) that covers the key features of concrete graph pattern languages and a first-order fragment of OCL (including expressions like `and`, `or`, `not`, `collect`, `select`, `exists`, `forall`, `includes`, `excludes`, but excluding expressions like `size`, `min` or `max`). In our current implementation, we used the graph pattern language of VIATRA [58,60], where error patterns describe malformedness of the model, and derived logic predicates in accordance with [47].

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

Semantics A graph predicate $\varphi(v_1, \ldots, v_n)$ can be evaluated on model M along a variable binding $Z : \{v_1, \ldots, v_n\} \to Obj_M$ from variables to objects in M. The truth value of φ can be evaluated over model M along the mapping Z (denoted by $[\![\varphi(v_1, \ldots, v_n)]\!]_Z^M$) in accordance with the semantic rules defined in Fig. 4.

If there is a variable binding Z where the predicate φ is evaluated to 1 over M (i.e., $[\![\varphi]\!]_Z^M = 1$) is often called a *pattern match*. Otherwise, if there are no bindings Z to satisfy a predicate, i.e., $[\![\varphi]\!]_Z^M = 0$ for all Z, then the predicate φ is evaluated to 0 over M. Graph query engines like [58] 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 denotes a violation, thus the corresponding graph formula needs to capture the erroneous case. In order to ensure validity, some of the predicates $\mathtt{P}_k \in \Sigma$ can be marked as WF constraints.

Example 4 Two WF constraints checked by the Yakindu environment can be captured by graph predicates of error patterns as follows:

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

2.4 Motivation: 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 preconditions 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 designing test suites for software artifacts, also applicable to testing DSL tools, is *mutation testing* [37,52]. Mutation testing aims to quantify the quality of test suites by (1) deriving a set of mutant artifacts (e.g., WF constraint sets in our case) by applying a set of mutation operators. Then, (2) the test suite is executed for the evaluation of both the original and the mutant artifacts (WF checkers), and (3) their outputs are compared. (4) A mutant is *killed* by a test if different output is produced for the two cases (i.e., different match sets); in other words if the test input is evidence for the different behaviors of the mutant and the original artifacts. (5) The mutation score of a test suite is calculated either as the number of mutants killed by at least one test (from a fixed collection of mutants), or the ratio of mutants killed wrt. the total number of mutants. A test suite with better mutation score is preferred [32].

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 $\varphi_e = \varphi_1 \wedge \cdots \wedge \varphi_k$, all unbound variables are quantified existentially ($\exists$), and may refer to other predicates specified in the same form. Note that this 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., $\varphi_e = [\varphi_1 \wedge \ldots \wedge \varphi_i \wedge \ldots \wedge \varphi_k]$ is rewritten to $\varphi_f = [\varphi_1 \wedge \cdots \wedge \varphi_{i-1} \wedge \varphi_{i+1} \wedge \cdots \wedge \varphi_k]$, or may forget a negation (Negation Omission), i.e., $\varphi_e = [\varphi_1 \wedge \ldots \wedge (\neg\varphi_i) \wedge \ldots \wedge \varphi_k]$ is rewritten to $\varphi_f = [\varphi_1 \wedge \ldots \wedge \varphi_i \wedge \ldots \wedge \varphi_k]$.

Given an instance model M, we assume that both $[\![\varphi_e]\!]^M$ and the faulty $[\![\varphi_f]\!]^M$ 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 5 According to our fault model, we can derive two mutants for *incomingToEntry* of Example 4 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 $[\![\varphi]\!]^{M_1}_Z = 0$ and $[\![\phi]\!]^{M_1}_Z = 0$ for all Z. As a test model, both M_1 and M_2 are 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.

2.5 Motivation: testing model-based access control policies

A second motivating case study is the testing of *rule-based*, *model-level* access control policies for *collaborative modeling*. In our setting, the design model of a complex system, hosted by a system integrator, must be partially accessible (for reading, and in some cases for writing) to multiple subcontractors, downstream supply chain actors, remote offices, certification authorities, etc., according to an access control policy set up by a policy engineer. Erroneous access control settings can have a critical impact [27], due to export control regulations, the high business value of intellectual property (IP) contained in the models, and the importance of adherence to change request procedures. However, in practice, it may be difficult to properly implement access control policies based on informal security requirements (see examples in [8,18,23]) and the interactions or conflicts between different access rules or requirements may not be well thought-out.

An example access rule may grant certain specialists full access to subsystems they own, as well as the contents of such subsystems. The specialists must also be able to read or write signals directly provided as outputs of control units transitively contained in these subsystems; this requires further access rules in the policy. As this example shows, the complexity of such rules may be quite high, and therefore properly formalizing them may be prone to errors.

Following earlier work including [9,27,35], we propose test generation for access control policies. In our case, this means the generation of instance models, on which access control policies can be evaluated and the results inspected, so that the policy engineer can verify that the policy indeed works as intended. As manual inspection (human-in-the-loop testing) is time consuming, *test prioritization* (the order in which the generated test cases are evaluated) is of great importance: The goal is for the test suite to reveal as many

bugs of the policy as possible, using as few test models as possible. This motivates the need for *ordering heuristics* for tests to support test selection / prioritization.

In our experiments, we rely on the policy language of the MONDO COLLABORATION FRAMEWORK [18], which is (i) *fine-grained* in the sense that each model element is assigned its own set of permissions; and (ii) *rule-based* with single rules granting or denying permissions for many elements in a model (selected according to an expressive *graph query* / predicate, see Sect. 2.3). A *MONDO Access Control Policy* combines individual rules with various priority classes, and ensures *referential integrity* [8] of the filtered view model throughout the interactions of rules.

The case study applies access control to models of the wind turbine domain, according to security requirements from the literature and evaluates generated (consistent and diverse) test models with respect to their ability to distinguish the original access control policy from its mutant policies. A detailed description of the security requirements, the baseline policy, the fault model, and the obtained mutants, is relegated to the online appendix [2].

Challenge Unlike (non-model-based) standard rule-based policy languages such as XACML [21], *MONDO Access Control* applies to *graph-structured models*, and uses expressive model queries (graph formulae) to identify the elements to which the rules of the policy are applicable. This presents the main challenge of this case study: Automated testing of model-based access control requires the generation of *consistent and diverse* (graph) models that demonstrate the behavior of the access rules.

This is in contrast with existing test generation and mutation testing approaches (e.g., [9,10,27,35]) for policy languages such as XACML, where test inputs usually feature a finite set of parameters for combinatorical or other test generation approaches to consider. Therefore, the novelty of our investigation is tied to the rich and expressive nature of graph models and is specifically focusing on the ability of graph model generators and measures of model diversity to efficiently test model-based access control policies.

3 Shape-based model diversity metrics

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 [41] divides the input space of a program into equivalence classes based on observable output, and then selects the different test cases of a test suite from different equivalence classes to achieve a diverse test suite. However, while software diversity has been studied extensively [7], model diversity is much less covered.

In existing approaches [11,12,14,15,46,59] for testing DSL and transformation tools, a test suite should provide full *metamodel coverage* [62], and it should also guarantee that any pairs of models in the test suite are non-isomorphic [31,56]. In [61], 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.

In this section, we propose diversity metrics for a single model, for pairs of models and for a set of models based on neighborhood shapes [43], a formal concept known from the state space exploration of graph transformation systems [42]. *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 Structural diversity of models

An effective test suite needs models with diverse graph structures. This paper proposes various metrics to measure the diversity of graph structures by adapting the formal concepts of graph shapes. The proposed distance metrics measure the diversity of a test suite based on the number of structural differences between models: If two models are the same, then the related distance metric equals to zero. Models are "far from each other" if the number of different structures is high, and they are close if the distance metric is low. Our intuition is that a more diverse model set wrt. our metrics will serve as a better test suite as it checks a larger number of different graph structures.

Example 6 Figure 5 collects the number of occurrences of four sample structures (subgraphs) in instance models illustrated in Fig. 1. First, both M_1, M_2 and M_3 contains an `Entry` with an outgoing `Transition` (Structure 1) once. However, M_3 contains also an `Entry` state with both incoming and outgoing `Transitions` (Structure 2), which can distinguish model M_3 from M_1 and M_2. Therefore, if a design flaw (like a missing well-formedness constraint) is characterized with this structure, it can be detected with model M_3, but not with models like M_1 and M_2.

Similarly, Structure 3 describes `States` with both incoming and outgoing `Transitions`, which occurs 2, 3 and 1 times in models M_1, M_2 and M_3, respectively. Structure 4 describes only incoming `Transitions`, and the

[2] https://github.com/FTSRG/publication-pages/wiki/Diversity-STTT-2019

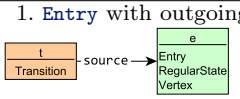
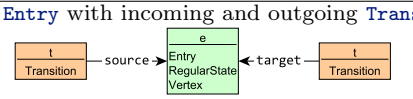
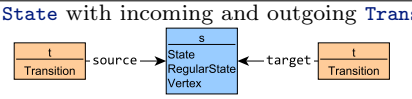
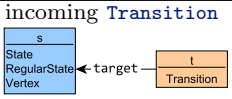
Structure	M_1	M_2	M_3
1. `Entry` with outgoing `Transition` 	×1	×1	×1
2. `Entry` with incoming and outgoing `Trans.` 	×0	×0	×1
3. `State` with incoming and outgoing `Trans.` 	×2	×3	×1
4. `State` with incoming `Transition` 	×2	×3	×2

Fig. 5 Number of different structures in models

occurrences of this structure increase to 2 in model M_3. This increase indicates a `State` without any outgoing `Transitions` (i.e., deadlocks), which may describe a challenging scenario. On the other hand, the number of occurrences of Structures 3 and 4 cannot distinguish models M_1 and M_2, which indicates that they are similar with respect to those structures.

However, the number of potential structures can be huge even for small graphs. To systematically count the occurrences of *all graph structures with a specific size* in a graph model, this paper uses *neighborhood shapes* [43], which categorizes all nodes of a graph based on their neighborhood for a given range, thus collecting all potential structures. The number of structural differences is calculated on the level of shapes (which are in between traditional metamodels and instance models), and such shapes are then used to quantify the distance between models.

3.2 Neighborhood shapes of graphs

A neighborhood describes the local properties of an object in a graph model for a range of size $i \in \mathbb{N}$. According to [43], the neighborhood describes the class and edge relations of an object. Intuitively, neighborhood shapes start splitting the classes and associations of a metamodel by introducing subtypes if the neighborhood of certain elements can be structurally different. For example, in case of statecharts, shaping may split the general `State` class into subclasses such as `InitialState`, `IntermediateState`, `TrapState`, etc. based on the existence or nonexistence of incoming and outgoing `source` and `target` edges.

Here, we propose an extension of the shape concept [43] to support hypergraphs (in other words, predicates/base relations of higher arity). Besides regular edge relationships (with objects at the source and target end of a given edge type), we also consider generalized hyperneighbor relationships (where such a hyperedge is defined by a match of a predicate with arbitrary arity leading between objects). Since classes and references can be regarded as special predicates (with arity 1 and 2), the proposed concept handles relations specified by any predicate $\mathtt{P} \in \Sigma$ uniformly.

Concretely, a neighborhood of range i for an object $o \in Obj_M$ contains descriptors of objects that can be reached from o by a path of at most i hyperedges. Technically, $nbh_i(o)$ (range i neighborhood of o) contains range $i-1$ neighborhoods of objects with a common hyperedge, which by recursion contain range $i-2$ neighborhoods of their neighbors (which are objects reachable from o in at most $i-2$ steps), and so on.

Definition 1 (*Generalized neighborhood*) Formally, neighborhood descriptors are defined recursively for a range i with the predicate symbols of Σ.

- As a base case, for range $i = 0$, the neighborhood is an empty set: $Nbh_0 = \emptyset$.
- For ranges $i > 0$ neighborhood descriptors are build upon the description of references Rel_i, which consists of predicate symbols $\mathtt{P} \in \Sigma$, parameter indexes of $\mathtt{P}$ and $\alpha(\mathtt{P}) - 1$ neighborhoods. Therefore, $Rel_i \subseteq \Sigma \times \mathbb{N} \times \bigcup_{\mathtt{P}\in\Sigma} Nbh_{i-1}^{\alpha(\mathtt{P})-1}$.
- For range $i > 0$ the neighborhood of an object can be formalized as $Nbh_i = Nbh_{i-1} \times 2^{Rel_i}$, where in a tuple $\langle n, r\rangle$ n is the neighborhood of the object with range $i-1$ and r describes the set of its relations.

The shaping function $nbh_i : Obj_M \to Nbh_i$ maps each object in a model M to a neighborhood with range i: (1) if $i = 0$, then $nbh_0(o) = \emptyset$; (2) if $i > 0$, then $nbh_i(o) = \langle nbh_{i-1}(o), rel\rangle$, where

$$
\begin{aligned}
rel = \{\langle \mathtt{P}, j, n_1, \ldots, n_{j-1}, n_{j+1}, \ldots, n_{\alpha(\mathtt{P})}\rangle \mid \\
\exists o_1, \ldots, o_{j-1}, o_{j+1}, \ldots, o_{\alpha(\mathtt{P})} \in Obj_M : \\
[\![\mathtt{P}(v_1, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_{\alpha(\mathtt{P})})]\!]^M_{v_1\mapsto o_1,\ldots,v_{\alpha(\mathtt{P})}\mapsto o_{\alpha(\mathtt{P})}} \wedge \\
\bigwedge_{\substack{0<k\le\alpha(\mathtt{P}),\\ k\neq j}} [n_k = nbh_{i-1}(o_k)]\}.
\end{aligned}
$$

A *(graph) shape* of a model M for range i (denoted as $S_i(M)$) is a multiset of neighborhood descriptors of the model: $S_i(M) = (Nbh_i, m_M)$ where $m_M : Nbh_i \to \mathbb{N}$ assigns multiplicities to the neighborhoods: $m_M(n) = |\{o \in Obj_M \mid nbh_i(o) = n\}|$. We will use the size of a shape $|S_i(M)|$ as the number of shapes used in M, i.e., $|S_i(M)| = |\{n \in Nbh_i | m_M(n) > 0\}|$.

Example 7 We illustrate the concept of graph shapes for model M_1 (visualized in Fig. 6). For range 0, all local neighborhood descriptors are the same: $\forall o \in M_1 : nbh_0(o) = \emptyset$.

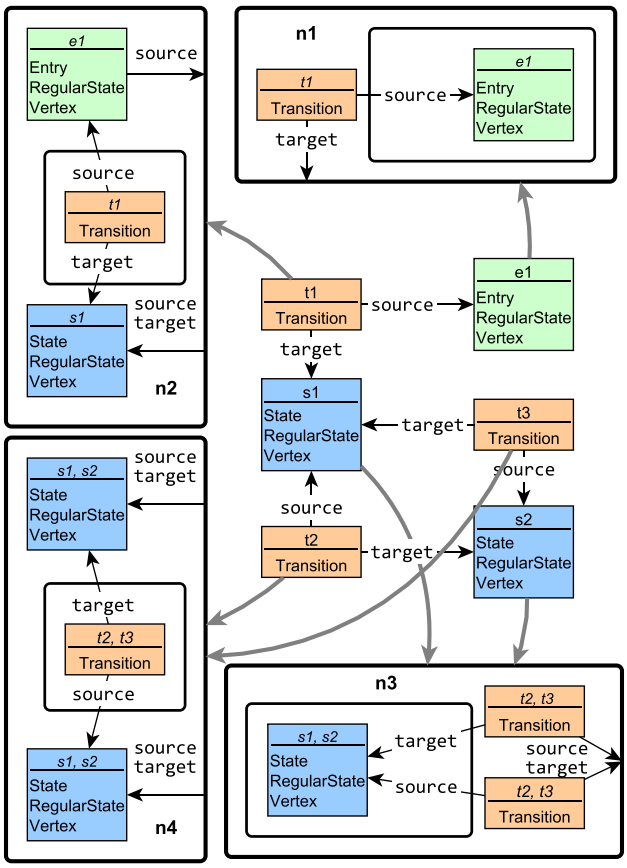


Fig. 6 Neighborhoods in M_1 for range 2

For range 1, objects are mapped to class and reference symbols as neighborhood descriptors (for lack of predicate matches of WF constraints):

$$nbh_1(e1) = \langle \emptyset, \{\langle \texttt{Entry}, 1\rangle, \langle \texttt{RegularState}, 1\rangle, \langle \texttt{Vertex}, 1\rangle, \langle \texttt{source}, 2, \emptyset\rangle\}\rangle$$

Which describes that for node $e1$, unary predicates `Entry`, `RegularState` and `Vertex` are true if $e1$ is substituted for the first (1) parameter, and binary predicate `source` is true in some cases (namely in $\texttt{source}(t1, e1)$) where node $e1$ is in the second (2) parameter of this predicate. However, for range 1, the neighborhood does not describes more information about the other node $t1$ (as illustrated with the internal box). The neighborhood of the other nodes is calculated similarly:

$$nbh_1(t1) = nbh_1(t2) = nbh_1(t3) = \langle \emptyset, \{\langle \texttt{Transition}, 1\rangle, \langle \texttt{source}, 1, \emptyset\rangle, \langle \texttt{target}, 1, \emptyset\rangle\}\rangle$$

And for the states:

$$nbh_1(s1) = nbh_1(s2) = \langle \emptyset, \{\langle \texttt{State}, 1\rangle, \langle \texttt{RegularState}, 1\rangle, \langle \texttt{Vertex}, 1\rangle, \langle \texttt{source}, 2, \emptyset\rangle, \langle \texttt{target}, 2, \emptyset\rangle\}\rangle$$

In this example, objects of the same class have the same neighborhood for range 1. Accordingly, we denote the (range 1) neighborhoods of `Transitions` as $nbh_1(t)$ and `States` as $nbh_1(s)$.

For range 2, objects are further split based on the types on the other ends of the references, 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 `States`.

$$nbh_2(e1) = \langle nbh_1(e1), \{\langle \texttt{Entry}, 1\rangle, \langle \texttt{RegularState}, 1\rangle, \langle \texttt{Vertex}, 1\rangle, \langle \texttt{source}, 2, nbh_1(t1)\rangle\}\rangle$$

$$nbh_2(t1) = \langle nbh_1(t), \{\langle \texttt{Transition}, 1\rangle, \langle \texttt{source}, 1, nbh_1(e1)\rangle, \langle \texttt{target}, 1, nbh_1(s)\rangle\}\rangle$$

$$nbh_2(t2) = nbh_2(t3) = \langle nbh_1(t), \{\langle \texttt{Transition}, 1\rangle, \langle \texttt{source}, 1, nbh_1(s)\rangle, \langle \texttt{target}, 1, nbh_1(s)\rangle\}\rangle$$

$$nbh_2(s1) = nbh_2(s2) = \langle nbh_1(s), \{\langle \texttt{State}, 1\rangle, \langle \texttt{RegularState}, 1\rangle, \langle \texttt{Vertex}, 1\rangle, \langle \texttt{source}, 2, nbh_1(t)\rangle, \langle \texttt{target}, 2, nbh_1(t)\rangle\}\rangle$$

The neighborhoods of range 2 are depicted in Fig. 6. For range 3, each object of M_1 would be mapped to a unique element.

Figure 7 next illustrates the construction of a type graph from neighborhoods of shape $S_1 = \{n_1, \ldots, n_4\}$ of the nodes of graph M_1. Here, boxes with $n_1, \ldots, n_4$ represent the group of nodes with the same shape for range 2, and edges between the boxes representing the references between the groups. In Fig. 8, the neighborhood shapes of models M_1, M_2, and M_3 for range 1 are represented in a visual notation adapted from [43,44] (without additional annotations e.g., multiplicities or predicates used for verification purposes). The trace of the concrete graph nodes to neighborhood is illustrated on the right. For instance, $e1$ and $e2$ in $M1$ and M_2 `Entries` are both mapped to the same neighborhood $n1$, while $e3$ can be distinguished from them as it has incoming reference from a transition, thus creating a different neighborhood $n5$.

Properties of graph shapes The theoretical foundations of graph shapes [43,44] 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., $|S_i(M)| \leq |S_{i+1}(M)|$.

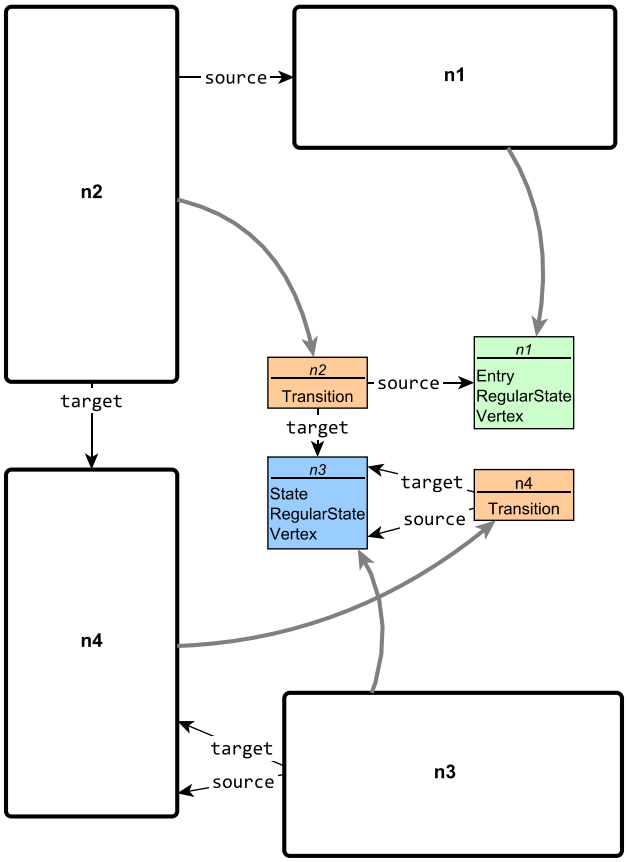


Fig. 7 Construction of type graph from neighborhoods n_1,…,n_4

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

3.3 Shape-based model diversity

We define two types of metrics for model diversity based upon neighborhood shapes. *Internal diversity* captures the diversity of a single model (or model set), i.e., it can be evaluated individually for each and every generated model. This model diversity metric measures the number of neighborhood types (object categories) used in the model (or model set) with respect to the size of the model(s). *External diversity* captures the *distance* between a pair of models, or the overall degree of variation in a larger set of models.

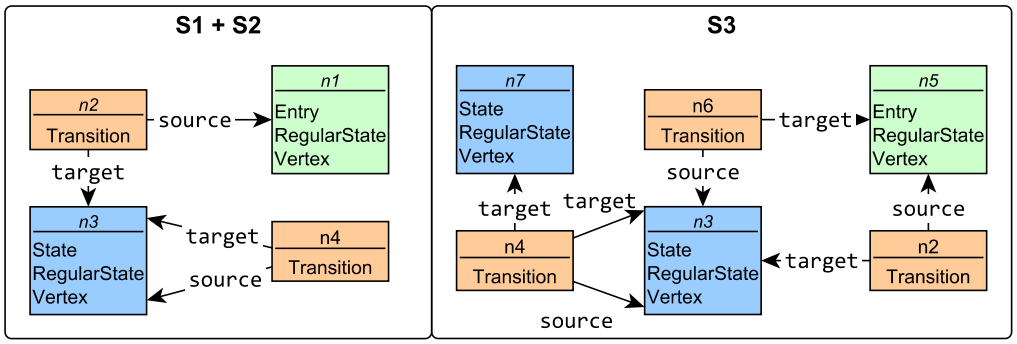
Internal model diversity for one model Internal model diversity measures the number of different neighborhood with respect to the number of nodes. The range of this internal diversity metric $d_i^{\text{int}}(M)$ is normalized to [0..1], and for a model M with $d_1^{\text{int}}(M) = 1$ ensures each object has some different property (i.e., there is a predicate that can differentiate between each node).

Definition 2 (*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: $d_i^{\text{int}}(M) = |S_i(M)|/|M|$.

This metric helps to minimize test inputs with respect to model size by punishing the unnecessary copying of the same model fragments (which is typical output for a logic solver). Models with higher internal diversity is frequently preferred in numerous machine learning or testing scenarios where a certain approach is more sensitive to the number of examples, but not insensitive to their size.

For simplicity, our definition for internal model diversity (as well as other metrics presented in this paper) takes the designated shape range as a parameter. However, it is possible to derive a generalized metric for internal model diversity that includes shapes of all ranges. For instance, the generalized metric for internal diversity can be defined as $d^{\text{int}}(M) = \sum_{i \in \mathbb{N}} \frac{d_i^{\text{int}}(M)}{2^{i+1}}$. In this formula, internal diversities of smaller ranges have larger coefficients because the differences in smaller neighborhoods are more significant. Other aggregation methods are also possible to use.

Coverage of model set The coverage of a model set measures the number of different shapes (for a given range) that appear in at least one of the models.



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. 8 Sample neighborhood shapes of M_1, M_2 and M_3

Definition 3 (*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 $cov_i\langle MS\rangle = |S_i(M_1) \cup \cdots \cup S_i(M_k)|$.

For a specific range i, the number of potential neighborhood shapes within that range is finite, but it grows superexponentially. As such, the coverage of a model set is not normalized, but its value grows monotonously for any range i by adding new models. This way, this is not yet an appropriate diversity metric for a set of models.

On the positive side, the proposed coverage concept $cov_i\langle MS\rangle$ generalizes metamodel coverage [20,62], which is a frequently used coverage metric. Metamodel coverage requires test input for each attribute (feature coverage), each subtype of a type (inheritance coverage) and references (potentially with multiple representative multiplicities, association coverage) of a metamodel. Compared to metamodel coverage, shape coverage requires test case for each *possible combination* of those features (for a specific range).

Internal diversity of model set In order to obtain a diversity metric for a set of models, we normalize the coverage by the size of the models as follows.

Definition 4 (*Diversity of model set*) For a range i of neighborhood shapes for model set $MS = \{M_1, M_2, \ldots, M_n\}$, the internal diversity of MS is the number of covered shapes proportional to the total (combined) size of the models: $d_i^{\text{int}}(MS) = cov_i\langle MS\rangle / \Sigma_j |M_j|$.

For a small range i of neighborhood shapes, one can derive a model M_j with $d_i^{\text{int}}(M_j) = 1$, but for larger models M_k (with $|M_k| > |M_j|$) we will likely have $d_i^{\text{int}}(M_j) \geq d_i^{\text{int}}(M_k)$. However, due to the rapid growth of the number of shapes for increasing range i, for most practical cases, $d_i^{\text{int}}(M_j)$ will converge to 1 if M_j is sufficiently diverse.

External model diversity Finally, we also wish to measure the structural diversity between two models, which is captured by the concept of external diversity which combines shaping with existing pseudo-distance metrics. To increase the generality of our definition, we make the actual pseudo-distance functional d to be a parameter, and we adapt concrete metrics to calculate distances in the context of shapes.

Definition 5 (*External model diversity*) A function $d : \mathcal{M} \times \mathcal{M} \to \mathbb{R}$ is an external model diversity function if *(i)* d is a pseudo-distance and *(ii)* the value of $d(M_j, M_k)$ can be calculated from $S_i(M_j)$ and $S_i(M_k)$.

3.4 Distance metrics for model diversity

Distance metrics characterize the difference between two models. For usability, it is required that a metric is a pseudo-distance over models in the mathematical sense [4], and thus, it can serve as a diversity metric for a model generator in accordance with [61].

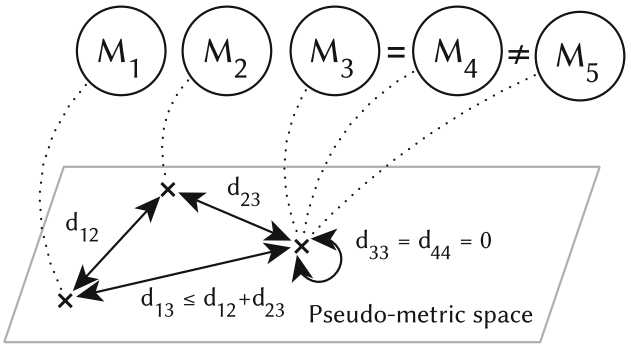


Fig. 9 Illustration of pseudo-distance

Definition 6 (*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: $d(M_j, M_l) \leq d(M_k, M_j) + d(M_j, M_l)$

Figure 9 illustrates the calculation of distances between $M_1, \ldots, M_5$ by mapping them to a pseudo-metric space. First, as in classic geometry, all distances are positive, and symmetric. For pseudo-distances, we assume triangle inequality, which means that direct distance (e.g., between models M_1 and M_3) is less or equal than the sum of indirect paths (like from M_1 to M_2 and from M_2 to M_3). Additionally, the mapping have to be functional, which means that if two models are isomorphic (like M_3 and M_4) it has to mapped to the same point. Moreover, even non-isomorphic models can be mapped to the same point (like M_4 and M_5). Therefore, $d(M_i, M_j) = 0$ is not implies that M_i and M_j isomorphic (just they are similar with respect to d). On the other hand, if $d(M_i, M_j) > 0$ implies that M_i and M_j are non-isomorphic.

The first metric considers the number of different shapes contained in the models:

Definition 7 (*Symmetric distance*) Given a range i of neighborhood shapes, the *symmetric distance* 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, $d_i^{\text{sym}}(M_j, M_k) = |\{n \in Nbh_i \mid m_{M_j}(n) > 0 \wedge m_{M_k}(n) = 0 \bigvee m_{M_j}(n) = 0 \wedge m_{M_k}(n) > 0\}|$.

The second distance metric is derived from the cosine of the angle between shape count vectors, where the dimensions are the local neighborhoods and the coordinates of the shapes are the corresponding multiplicities:

Definition 8 (*Cosine distance*) Given a range i of neighborhood shapes, the *cosine distance* $d_i^{\cos}(M_j, M_k)$ of models M_j and M_k represents an angular distance between two shapes. The value of $d_i^{\cos}(M_j, M_k)$ is derived from the angle between the shape vectors.

$d_i^{\cos}(M_j, M_k) = 1 - \cos_i(M_j, M_k)$, where $\cos_i(M_j, M_k)$ is the *cosine similarity* of the shape vectors:

$$\cos_i(M_j, M_k) = \frac{\sum_{n \in Nbh_i} m_{M_j}(n) m_{M_k}(n)}{\sqrt{\sum_{n \in Nbh_i} (m_{M_j}(n))^2} \sqrt{\sum_{n \in Nbh_i} (m_{M_k}(n))^2}}$$

Technically, the presented definition does not fit Definition 5, because it does not satisfy the *triangle inequality* property of Definition 6. However, since the exceptions are rather extreme cases, this formula is widely used as a *semi-metric*.

During model generation, we will exclude a model M_k if $d_i^{\text{sym}}(M_j, M_k) = 0$ 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 distance metrics can be considered a dual of symmetry-breaking predicates [56] 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).

Example 8 Let us calculate the different diversity metrics for M_1, M_2 and M_3 of Fig. 1. For range 1, they have the shapes illustrated in Fig. 8. The internal diversity of those models are $d_1^{\text{int}}(M_1) = 4/6$, $d_1^{\text{int}}(M_2) = 4/8$ and $d_1^{\text{int}}(M_3) = 6/7$, thus M_3 is the most diverse model among them. As M_1 and M_2 has the same shape, the symmetric distance between them is $d_1^{\text{sym}}(M_1, M2) = 0$. The distance between M_1 and M_3 is $d_1^{\text{sym}}(M_1, M3) = 4$ as M_1 has 1 different neighborhoods ($n1$), 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 neighborhoods ($n1$–$n7$). □

3.5 Diversity-based model ordering for test case prioritization

In addition to test input efficiency, the efficiency of a test suite also matters: It needs to detects faults using as few individual tests as possible - especially if the automatically generated test input graphs are to be complemented with expected outputs by a human oracle. As motivated earlier, given a fixed set of graph models, we need to be able to select a small subset to be actually used as test inputs, so that they are still likely to detect a large number of faults. This requires diversity-based *ordering heuristics* in support of test prioritization / selection.

Based on the introduced diversity metrics, we propose several model orderings $(M_1, M_2, \ldots)$ of a given set of models that focus on maximizing the diversity of prefix sequences $(M_1, \ldots, M_i)$ in order (informally, the models are sorted based on *most diverse first*).

The first ordering is based on the internal diversity of models and therefore can be calculated efficiently.

Definition 9 (*Coverage order*) Given a set of models MS, an order $(M_1, M_2, \ldots)$ is a coverage order of MS if *(i)* M_1 is of maximal shape: $\forall i \leq |MS| : |S(M_i)| \leq |S(M_1)|$ and *(ii)* for $i > 1$ M_i maximizes the coverage of $(M_1, \ldots, M_i)$: $\forall j : i \leq j \leq |MS| \rightarrow cov\langle M_1, \ldots, M_{i-1}, M_j\rangle \leq cov\langle M_1, \ldots, M_i\rangle$.

The following orderings are based on external diversity. In the following definition, $dist(M, M')$ can be replaced by any distance metric.

Definition 10 (*Distance order*) Given a set of models MS, an order $(M_1, M_2, \ldots)$ is a distance order of MS if *(i)* $dist(M_1, M_2)$ is the maximal distance in MS: $\forall j, k \leq |MS| : dist(M_j, M_k) \leq dist(M_1, M_2)$ and *(ii)* for $i > 2$ M_i maximizes the distance from $\{M_1, \ldots, M_{i-1}\}$: $\forall i \leq j \leq |MS| : \min_{k<i} dist(M_k, M_j) \leq \min_{k<i} dist(M_k, M_i)$.

Calculating distance order is less efficient as it requires all distances to be known from the beginning and therefore has both time and space complexity of $\mathcal{O}(|MS|^2)$. The space complexity can be reduced to $\mathcal{O}(|MS|)$ by omitting condition *(i)* and chosing the first element randomly or similarly to coverage order. This way the distances can be calculated on the fly.

Definition 11 (*Weak distance order*) Given a set of models MS, an order $(M_1, M_2, \ldots)$ is a weak distance order of MS if *(i)* M_1 is of maximal shape: $\forall i \leq |MS| : |S(M_i)| \leq |S(M_1)|$ and *(ii)* for $i > 1$ M_i maximizes the distance from $\{M_1, \ldots, M_{i-1}\}$: $\forall i \leq j \leq |MS| : \min_{k<i} dist(M_k, M_j) \leq \min_{k<i} dist(M_k, M_i)$.

3.6 From instance model to shaping

Section 3.2 defines how shapes are computed from a set of relations over the graph nodes; then, Sect. 3.4 introduces diversity and distance metrics, while Sect. 3.5 proposes orderings of test cases, all derived from the shaping. However, so far we have not specified how a given instance model is to be interpreted as a set of relations for the purposes of computing the shapes.

The straightforward solution is to take a unary relation for each class / node type, and a binary relation for each edge / reference type (ignoring attributes for conciseness). This results in classic *neighborhood-based* graph shaping [43].

However, this is not the only possibility. Instead of actual classes and edges, we may take an arbitrary set of relations (each identified by a predicate symbol $P \in \Sigma$) over the graph nodes that characterize the graph model according to some properties of interest. The definitions given in Sect. 3.2 are general enough so that they can be applied to these relations instead of the graph directly, resulting in a different set of node shapes. We call this *predicate-based* (or TVLA-style [44]) shaping. The two approaches can even be combined; it is possible to take both the basic graph structure and additional predicates to jointly determine shapes.

To motivate predicate-based shaping, let us discuss an intended use case. The literature on testing distinguishes *black-box* testing, where the artifact under test is not known at the time of test development (only its specification), from *white-box* testing, where the actual realization of the AUT can be taken into account to better focus the effort. Adapting this general terminology to testing a tool that processes models in certain modeling language, we find that a black-box test has to be developed with knowledge of the input DSL only; whereas a white-box approach is aware of how the tool processes its input models, e.g., which model queries (graph predicates / patterns) are used internally.

For example, consider the case of testing access control policies, as introduced in Sect. 2.5. For compiling a test suite or measuring its efficiency, a black-box testing approach would only consult the modeling language (the Wind Turbine domain) and aim for a diverse selection of model elements, whereas a white-box approach would additionally inspect the access control policy under consideration. Such a white-box technique may measure some notion of *coverage* of the policy provided by a given test suite, for example the proportion of access rules that applied (without being overruled) to at least one test case. Gaps in coverage would signal that more tests have to be developed or selected, as arbitrarily severe bugs may lurk in the untested parts of the policy; this is clearly useful for test selection/prioritization. Additionally, a white-box test input generator may aim to specifically generate such test cases that increase coverage. For example, the generator may take the graph patterns (predicates) used as access rule preconditions, and try to satisfy various combinations of these predicates. A similar white-box test generation approach was used for access control policies, e.g., in [35].

The above proposed generalized neighborhood-based shaping procedure can easily incorporate such additional white-box knowledge in the form of extra predicates (similarly to the *classifying terms* proposed in [22,26]). The additional predicates discussed in Sect. 2.2 allow the definition of domain-specific graph predicates. Specifying such additional graph predicates for each graph pattern used in the AUT drives the graph generator to find models that are diverse in the described domain-specific aspect, as well as the local neighborhhod aspects. This way the proposed diversity metrics also guarantee model diversity from the point of view of the AUT, and may reward test cases and test suites that specifically satisfy or violate relevant graph predicates in several combinations.

4 Evaluation

So far, we have proposed diversity-related properties of graphs and sets of graphs that we hypothetize to be useful for predicting or improving the quality and efficiency of test suites formed by automated generation of graph models. (Here, by "quality" of test suites, we mean their ability to detect faults in a modeling artifact; while "efficiency" also means that they do so in a small number of tests.) In this section, we will experimentally investigate the veracity of this claim. The experiments will apply mutation testing and use the obtained mutation scores as proxy for test suite quality. In short, we provide an experimental evaluation of our diversity metrics, based on mutation testing, with respect to various model generation techniques.

We address the following research questions:

RQ1: How effective are different external diversity metrics for test selection/prioritization in improving the overall mutation score for a test suite?
RQ2: How effective are different model generation techniques for test input generation in mutation testing?
RQ3: How effective is the internal diversity metric in predicting the mutation score of an individual instance model as test input?

4.1 Target domains and artifacts under test

In order to answer those questions, we executed model generation campaigns in our two case studies.

For both case studies, the campaigns involved generating a large set of test input models (conforming to the modeling language used in the case study), using multiple different model generation mechanisms. The generated models were evaluated both "statically" with respect to diversity metrics defined earlier, and also "dynamically" with regards to their quality as test inputs to detect faults in the AUTs specific to the case study. The latter involved applying an appropriate fault model and injecting corresponding faults into the AUT to obtain a large number of mutant artifacts, and then evaluating whether the test input models are able to differentiate the mutants from the original AUT. Beside the total number of mutants killed by a test suite, we also investigated how this mutation score increases with each additional test (according to a given ordering of graphs), in order to find test selection approaches that reach high test suite quality using a small number of tests.

Table 1 Case studies overview

Case Study	ST	WT
DSL style	Behavioral	Structural
AUT	WF constraints	Access policy
Inputs versus WF	Arbitrary	Consistent
Mutated	Patterns	Everything else
Extra predicates	– (black-box)	Optional

An overview of case study-specific details follows in the next paragraphs, with the most important differences highlighted in Table 1.

Yakindu Statecharts (**ST**) First, we used a DSL extracted from Yakindu Statecharts as proposed in [51]. We used the partial metamodel describing the state hierarchy and transitions of statecharts (illustrated in Fig. 2, 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. These WF constraints serve as AUTs (*not* "extra predicates" for white-box testing); i.e., we measured the ability of model generators to reveal bugs in WF constraints.

For mutation testing of WF constraints, we used a constraint or negation omission operator (CO and NO) to inject a fault 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 evaluated both the original and mutated versions of the constraints for each instance model. 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 mutant WF constraints killed that way.

Since the WF constraint is suspicious (not assumed to be correct) in this case study, the model generators were *not* instructed to obey them (i.e., consistency was not required), as the goal of generated models is to reveal discrepancies in WF.

Wind Turbine control system models (**WT**) As a second case study (which is novel compared to [50], and described in more detail in the online appendix [3]), we used WT system models conforming to the metamodel illustrated in Fig. 3, containing 15 classes, 18 references, two relevant attributes, which is extended by two classes and an enum type to encode access control metadata. For this case study, we also formalized 10 WF constraints. Additionally, we formalized 25 access control rules using a further 18 graph patterns to define a complex access control policy motivated by the literature. This access control policy serves as the AUT; i.e., we measured the ability of model generators to reveal bugs in the policy.

For mutation testing of access control rules (**WT**), we derived 174 mutant policies by applying mutation operators on the access control rules. However, in **WT** mutations did not applied on the WF constraints. Moreover, mutations were allowed to change neither the 18 graph patterns underlying the policy, nor the way they are used in the access rules ("bindings"). This choice was deliberately made to reduce the overlap with the first case study; in **ST**, mutation affected graph patterns / model queries exclusively, while in **WT**, mutation affected all parts of the access policy except the model queries. This way our experiments can attest that the test generation methods can be used to verify both graph patterns and other kinds of DSL artifacts.

For evaluation, we applied both the original and the mutant access control policies on a model, and checked all read and write permission for all users and all model elements. A test input *model kills a mutant policy*, if there is a difference for any user in any read or write permission in any part of the model, compared to the effective permissions assigned by the baseline policy. The *mutation score for a test suite* is the total number of mutant policies killed that way.

Since the WF constraints are assumed to be correct in this case study, the model generators (where supported) were instructed to obey them (consistent model generation), to reveal real problems with the policy.

As explained in Sect. 3.6, in case of white-box testing, the queries underlying the access rules can be used to enhance the shaping function. Such extra predicates may potentially result in diversity metrics and orderings that reward diversity in triggering access rules (over diversity in graph structure).

4.2 Compared model generation approaches

Our test input models were taken from multiple sources.

- First, we generated models with our VIATRA Solver [48] (denoted with **VS**), a state-of-the-art scalable graph model generator. For **WT** case study, we also configured the generator to satisfy WF constraints, so that all generated models are consistent. We used the generator in three configurations:
  - **VS/All**: This configuration is able to generate all consistent models and non-isomorphic model for a specific scope (number of objects).
  - **VS+ i**: This incremental configuration reuses the search space between each model generation steps. The solver is configured to introduce at least 3 new node shapes in each steps, using neighborhood abstraction for range 1.

[3] https://github.com/FTSRG/publication-pages/wiki/Diversity-STTT-2019

  - **VS-i**: After finding a valid model, this non-incremental configuration restarts the search. Similarly, the solver is configured to introduce at least 3 new node shapes in each steps, using neighborhood abstraction for range 1.

– Next, we generated models for the same domains using the Alloy Analyzer [56], a well-known SAT-based relational model finder, used as a model generator for several model generation scenarios. For representing EMF metamodels, we used traditional encoding techniques [13,47]. We used the latest 4.2 build for Alloy with the default Sat4j [34] as back-end solver. We artificially prevented the deterministic run of the solver to enable statistical analysis by adding a random amount of additional true statements. To enforce model diversity, Alloy was configured with two different setups for symmetry-breaking predicates. All other configuration options were set to default.
  - **A/20**: default configuration. For greater values, the tool produced the same set of models.
  - **A/0**: minimal value for symmetry breaking. Between 0 and 20, lower values produced better models with respect to mutation scores.
– We used the *EMF random instantiator* [5] (**REMF**) to produce random models for the domains. **REMF** does not enforce WF constraints (i.e., the generated models will typically violate some of the constraints), and it produces models which are at least as large as the designated specified size (so the designated size is the minimum size).
– Finally, for the **ST** case study, we also 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 [61] representing real models which were *not* intentionally created for testing purposes. We did not have such manually created models for the **WT** case study.

4.3 Measurement setup

To address **RQ1**, we generated two sets of models for each case study:

– **VS/All**: We generated the complete set of models of a certain size using **VS** in the order the generator produces the models (this default ordering of models is denoted by **VS/Solver**). For case study **WT**, we generated all 3483 well-formed models with five objects; for case study **ST**, we generated all 4606 models with seven objects. Therefore, in this measurement, all non-isomorphic models are available to fairly evaluate the efficiency of different metrics.
– **REMF**: We generated the same number of models of same size using **REMF** as random generator. **REMF** does not guarantee the well-formedness of the models, and isomorphic models may be constructed multiple times.

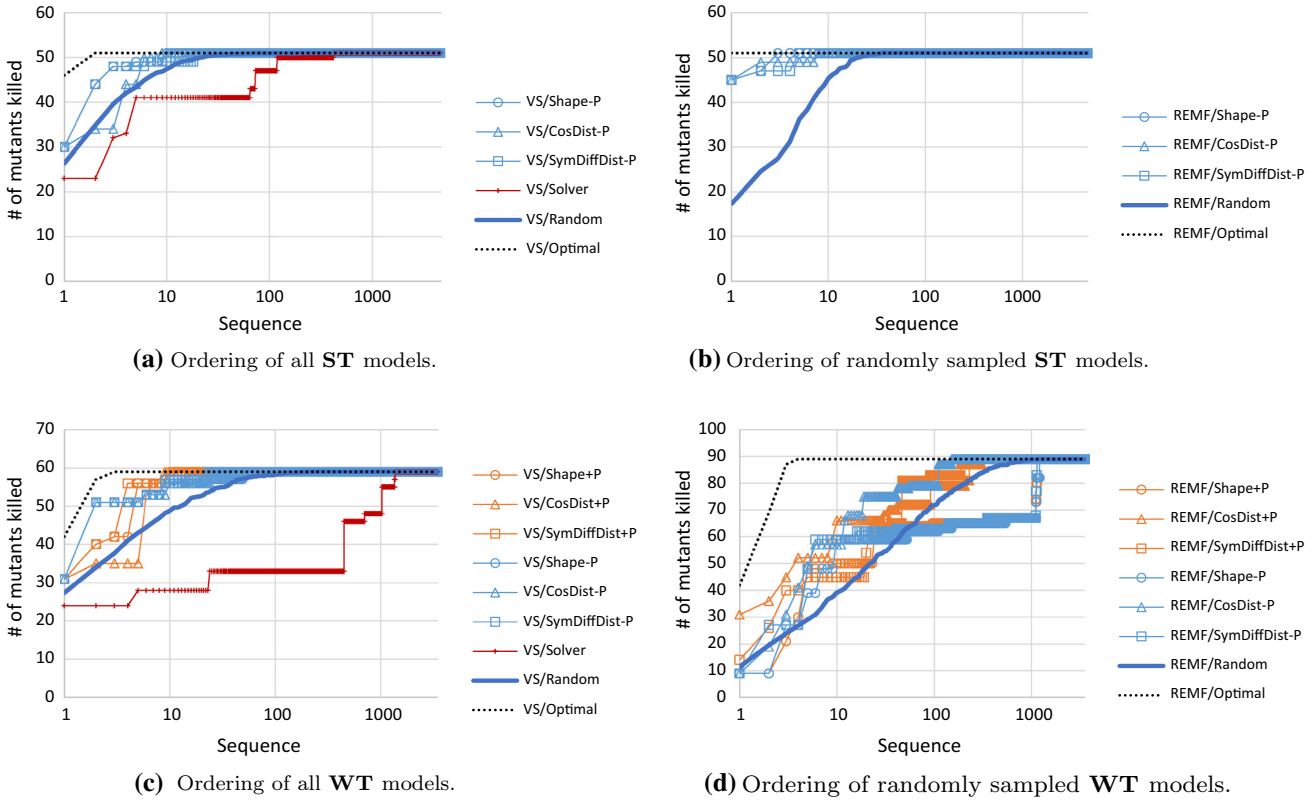
After generating the model set, we compared the efficiency of different ordering (test case prioritization) techniques on the model set, and measured the mutation score of the first n elements in the sequence.

To answer **RQ2** and **RQ3**, we introduced the following measurement setup. First, a sequence of instance models is generated with all **VS**, **A** and **REMF** 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 6000 models are generated for all configuration of **VS** , **A**and **REMF**, for both case studies). The target model size is uniformly set to 30 objects (as Alloy did not scale with increasing size). The size of **Human** models ranges from 50 to 200 objects. Next, we evaluate the mutation scores for all the models individually, for each prefix of the entire model sequence as test suites.

4.4 RQ1: external diversity for test prioritization

In order to answer **RQ1**, we checked the efficiency of all proposed orderings on test selection, both case studies (**WT** and **ST**) are executed on a complete set of instance models (created by **VS/All**) and a randomly generated set (generated by **REMF**). Figure 10 illustrates the efficiency of the various test case orderings for each tool and case study: **WT** and **ST** are illustrated top and bottom, **VS/All** and **REMF** are illustrated left to right. On a diagram, the horizontal axis shows the number of tests selected in a test suite (in log scale), in other words the length of the prefix of the complete model sequence. The vertical axis shows the achieved joint mutation score (a number of mutant killed by the test suite as a whole). Each chart line represents the mutation score achieved by a specific test case prioritization / ordering scheme, in context of the given tool and given case study.

– **Shape**, **CosDist**, **SymmDiffDist**, illustrate orderings with respect to Coverage order, Distance order with respect to Cosine distance, and Distance order with respect to Symmetric Difference.
– $+$**P** and $-$**P** denotes whether or not the shape abstraction aggregates *extra predicates* (see Sect. 3.6) in addition to the graph structure. **+P** is available only in case study **WT**, where such extra predicates represent the graph patterns underlying the access control rules (white-box testing).



(a) Ordering of all **ST** models.

(b) Ordering of randomly sampled **ST** models.

(c) Ordering of all **WT** models.

(d) Ordering of randomly sampled **WT** models.

Fig. 10 Mutation score for different ordering of model sequences (illustrated in log scale)

- **Solver** denotes the unchanged order in which **VS** originally produced the models.
- **Random** shows the average score of 50 random shufflings of the test suite.
- **Optimal** shows an optimal ordering of test models with respect to mutation score; i.e., for each test case, we select the model that increases the score of the test suite the most.

In both case studies, random order produced logarithmic characteristic in mutation score before killing all mutants. **VS** discovered mutants in multiple steps, stagnating for a large amount of models before increasing the mutation score again. Table 2 highlights when each ordering reached 95% and 100% of the maximal number of mutants it kills. For each case study **ST** and **WT** executed on either a complete set or a randomly sampled set of models, almost all proposed orderings performed significantly better than random ordering.

- Random ordering was always better than the order which **VS+i** produced the models. This is in accordance with the experience reported in [31].
- Surprisingly, complete mutation coverage could be achieved very quickly with optimal ordering (1–4 models). However, with the exception of the random sampling example of **WT**, at least one of our metrics was also able to provide complete coverage with fewer than ten models.
- Finally, in all cases, there was at least one test ordering heuristic which significantly outperformed both the original order and the random order, using one or two orders of magnitude fewer models.

Findings We can summarize our important findings of concerning reordering of models (**RQ1**):

> *By using the proposed external shape-based diversity metrics as heuristics for test prioritization, the same coverage can be achieved by using one or two orders of magnitude less models compared to random ordering or default model sequences derived by existing generators.*

However, there were no external diversity metrics that always produced better orderings than others; thus, we cannot propose a single best heuristic.

4.5 RQ2: test input generation techniques.

Figure 11 shows the number of killed mutants (vertical axis) by an increasingly longer sequence of models (horizontal

Table 2 Mutation score by order

Case study	Test suite	Order	95%	100%
ST		Shape-P	7	19
	All	CosDist-P	6	9
		SymDiffDist-P	7	19
	(4606 models)	Solver	118	420
		Random	12	202
		Optimal	2	2
	Randomly	Shape-P	3	3
	generated	CosDist-P	2	8
		SymDiffDist-P	5	5
	(4700 models)	Random	16	36
		Optimal	1	1
WT		Shape+P	10	10
		CosDist+P	9	9
		SymDiffDist+P	16	16
	All	Shape-P	9	50
		CosDist-P	25	25
	(3480 models)	SymDiffDist-P	23	23
		Solver	1363	1381
		Random	39	208
		Optimal	2	3
		Shape+P	1135	1135
		CosDist+P	230	308
	Randomly	SymDiffDist+P	118	262
	generated	Shape-P	1206	1206
		CosDist-P	111	151
	(3500 models)	SymDiffDist-P	1129	1129
		Random	330	1196
		Optimal	3	4

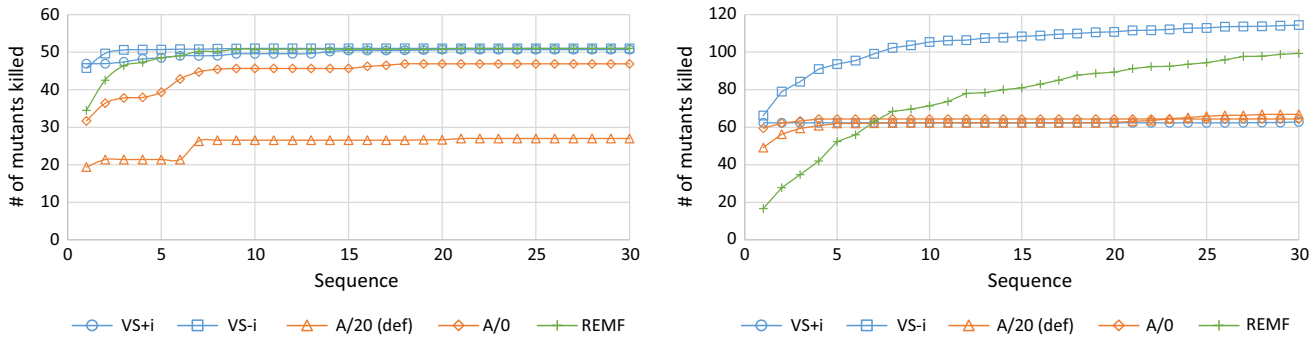
For case study **ST**, both **VS+i** and **VS-i** found a large amount of mutants in the first model, and the number of killed mutants (45+) which increased to 51. For **A**, the result highly depends on the symmetry value: for **s = 0** it found a large amount of mutants, but the value saturated early. The default configuration (**s=20**) found the least number of mutants.

For case study **WT**, **VS+i** and both **A** solvers started from relatively high mutation score at the beginning, but not a single run was unable to discover more than 75 bugs. At first **VS+i** showed better results, but at the end they showed similar efficiency. On the other hand **REMF** showed poor mutation score values at the beginning, but it was able to improve the score with each additional model, outpacing both **VS+i** and **A** after 7 models. Finally, **VS-i** showed the benefit of both **VS+i** and **REMF**: it produced relatively high mutation score at the beginning, and it was able to increase the mutation score in every step.

- From all generator **VS-i** produced the best mutation score, and combined the high initial mutation score of **VS+i** with the steepness of **REMF**.
- **REMF** provided good mutation score, but it dominantly generated malformed instances.
- From the three generators, **A** found the least amount of mutant.
- The significant difference in **VS-i** and **VS+i** in **WT** shows that restarting the solver had larger effect on the diversity heuristic (at least at the early stage).

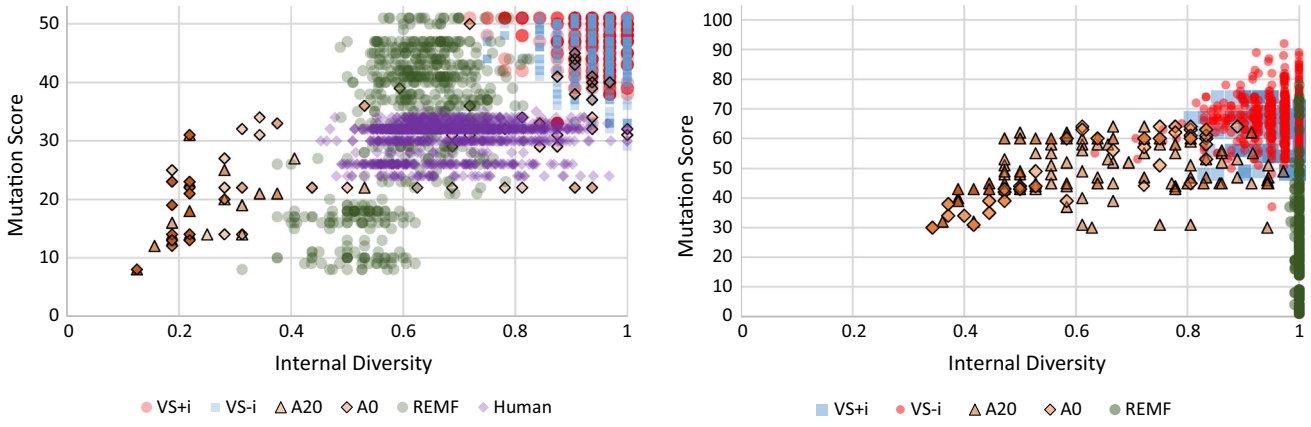
Findings Our important findings about the efficiency of existing model generators are the following (**RQ2**):

axis) generated by the different approaches. The diagram shows the *average* of 20 generation runs.



(a) Mutation Score of **ST** models. **(b)** Mutation Score of **WT** models.

Fig. 11 Mutation score of model sequences generated by multiple approaches



(a) Scatterplot of **ST** models. **(b)** Scatterplot of **WT** models.

Fig. 12 Correlation of model diversity and mutation score correlation

Table 3 Correlation values of internal diversity and mutation score

Model set	ST	WT
Autogenerated WF models	0.95	0.89
VS	− 0.01	− 0.05
A	0.55	0.73
REMF	0.54	− 0.16
Human	0.12	

*Graph solvers with random restarts (**VS-i**) and random graph generators (**REMF**) provided high mutation scores, where the former generated consistent instance models while the latter had better runtime performance. Since inconsistent models may not be loaded into modeling tools, logic solvers with random restart would be preferred to random graph generators in most testing scenarios. Deterministic model generators based on incremental graph solver (**VS+i**) or the logic solver (like Alloy, **A**) were significantly outperformed by randomized techniques. Altogether, effective model generators used for test input generation are randomized but consistent.*

4.6 RQ3: measurement results and analysis

Figure 12 illustrates the correlation between mutation score (horizontal axis) and internal diversity (vertical axis) for all generated models with 30 elements (1200 **A**, 1200 **VS**, 600 **REMF** for both case studies), and 1250 human models for **ST**.

Table 3 shows the correlation between mutation score and internal diversity for a group of models. We detected a high correlation on the set of well-formed models:

- On both case studies, automatically generated valid model sets (**A/0**, **A/20**, **VS-i**, **VS+i** and **REMF** in case of **ST**) showed high correlation of 0.95 and 0.89.
- **VS-i** and **VS+i** on itself does not show high correlation as all models have high diversity and mutation score.
- On the other hand, **A** generated both highly symmetric and diverse models, and internal diversity showed fairly good correlation in both case study (0.55 and 0.73).
- **REMF** produced mixed results: for **ST** it showed fairly good correlation, but in case of **WT** there was no correlation: all model had high diversity, but the mutation score was fairly low. This is because **WT** case study expected well-formed models, and **REMF** models was mostly ill-formed.
- Finally, **Human** models showed low correlation between mutation score and diversity, and it seams that those properties were independent.
- Generally speaking, **A** produced the lowest, **Human** and **REMF** medium, and **VS** high mutation score and diversity for single models.

Findings We highlight the following findings regarding the correlation between internal diversity and mutation score (**RQ3**):

High internal diversity indicates (correlates with) high mutation score; thus, our metrics can potentially be good predictors of the effectiveness of test suite for mutation testing of modeling tools.

However, there are certain model groups that showed no correlation. In particular, **REMF** was able to produce models with very high diversity, but with low mutation score. This can be partially considered to the fact that **REMF** models were mostly malformed.

Additionally, based on our measurement conducted on a large number of instance models created by humans, we conclude as follows:

Real instance models, created by humans with the intention to solve an engineering problem, provide low mutation score, thus they are ineffective for testing purposes.

4.7 Threats to validity and limitations

We evaluated more than 23,439 test inputs in our measurement, and all models were taken from two open-source industrial tools and for two different testing scenarios. Those case studies used wide range of metamodel and constraint features; thus, we did not specialized the generators to those case studies. Thus, we believe that similar results would be obtained for other domains.

For mutation operations, we checked only omission of predicates for WF, 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. For mutation testing of **WT**, we used mutation operators similar to [10,35]. In this paper, we focused on testing of structural constraints that can be captured as logic predicates (as detailed in Sect. 2.3).

Finally, although we detected a strong correlation between diversity and mutation score with our well-formed 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 **VS** or **A**, a higher diversity means a higher mutation score with high probability.

5 Related work

There are several initiatives to systematically cover the functionality of a modeling tool. In [20,62], modeling tools are specified as application using models as input, and the coverage critery is defined in MOF [28] level. It specifies the generic goal to create a set of test cases where at least one instance for every metamodel element is included.

Stochastic sampling and generation An advanced model generation approach was presented in the Formula Framework [30] using the Z3 SMT solver [17]. It is similar to our approach in the sense that both approaches are based on constraint solving. On the other hand, the approach of [31] applies stochastic random sampling to achieve a diverse set of generated models: In our work we rely more on the logical structure of the models to ensure diversity. Stochastic simulation is proposed for graph transformation systems in [57], 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.

[1] proposed a fuzzy logic-based model refinement framework that derives multiple test cases by extending simple models. In comparison, our proposed graph generation technique [48] uses similar principles, but with three-valued logic and partial model refinement instead of fuzzy logic. It may partially explain the quality of the generated models.

Mutation-based testing Diverse model generation plays a key role in testing model transformations code generators and complete development environments [40]. Mutation-based approaches [3,16,37] take existing artifacts (model transformations) and make random changes on them by applying mutation rules. A similar random model generator is used for experimentation purposes in [6]. Other automated techniques [12,19,45] generate models that only conform to the metamodel. While these techniques scale well for larger models, there is no guarantee whether the mutated models are well-formed.

White-box testing There is a wide set of model generation techniques which provide certain promises for test effectiveness. White-box approaches [2,3,11,24,25,46,47] 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 [33,51]. 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.

[22,26] proposes a similar predicate abstraction-based diversity and coverage metrics using OCL expressions: A set of relevant (unary) graph predicates are selected, and the logic solver aims to cover all combinations of possible evaluations of those predicates (e.g., for predicates p_1 and p_2 it tries to find models $M_1, \ldots, M_4$ where $\langle [\![M_i]\!]^{p_1}, [\![M_i]\!]^{p_2} \rangle \in \{\langle 0,0\rangle \ldots \langle 1,1\rangle\}$). As a difference, we use graph patterns which generalizes this technique to n-ary predicates. More importantly, our technique is able to automatically derive a large range of graph predicates from a neighborhood, thereby enabling the use of black-box testing scenarios.

Black-box testing Black-box approaches [13,20,25,38] 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 [31] aims to improve the diversity of generated models by taking exactly one element from each equivalence class defined by graph isomorphism, which can be too restrictive for coverage purposes.

6 Conclusion and future work

We proposed novel diversity and distance metrics for models based on shape analysis [43,44], which are true generalizations of metamodel coverage and graph isomorphism used in many research papers. We have shown how to incorporate additional information, e.g., for white-box testing. Based on these metrics, we have proposed test case prioritization schemes.

We evaluated our approach in a mutation testing scenario for two industrial case studies, for two different modeling domains and artifacts under test, and using four different sources of test input models.

We have found that among automatically generated consistent models, our internal diversity metric can predict the quality of individual models as test inputs. Additionally, we have shown that our test case prioritization heuristics generally outperformed a randomly ordered test suite (consisting of the same models) in terms of test quality. Finally, we have shown that an open-source graph solver [48], equipped with a (restart-based) iterative model generation strategy that aims to increase the proposed shape coverage metric, will outperform other model generation approaches. Note, however, that this effect is not observed with the incremental mode of the same generator, where locality effects annihilate the benefits of the proposed diversity.

While Alloy has been used as a model generator for numerous functional testing scenarios of DSL tools and model transformations [11,13,51,52,59], our measurements strongly indicate that it is not a justified choice as (1) the diversity and mutation score of generated models are problematic and (2) Alloy is very sensitive to configurations of symmetry-breaking predicates.

Although the metric-based iterative model generation scheme was shown to successfully produce diverse test suites, it remains to be seen whether deeper integration of shape-based metrics into the decisions of a graph solver is feasible and conductive to efficiently producing even higher-quality diverse test input sequences.

Acknowledgements Open access funding provided by Budapest University of Technology and Economics (BME). We would like to thank Tímea Balogh and Csaba Debreceni for providing us with their access control policy evaluator, to Aren Babikian, Boqi Chen, Chuning Li and Zoltán Micskei for their detailed review of the paper, as well as the anonymous reviewers for their constructive criticism that contributed to a better explanation of our setting.

References

1. Al-Refai, M., Cazzola, W., Ghosh. S.: A fuzzy logic based approach for model-based regression test selection. In: MoDELS, pp. 55–62. IEEE (2017)
2. Almendros-Jiménez, J.M., Becerra-Terón, A.: Automatic generation of ecore models for testing ATL transformations. In: MEDI, volume 9893 of LNCS, pp. 16–30. Springer (2016)
3. Aranega, V., Mottu, J., Etien, A., Degueule, T., Baudry, B., Dekeyser, J.: Towards an automation of the mutation analysis dedicated to model transformation. Softw. Test. Verif. Reliab. **25**, 653–683 (2015)
4. Arkhangel'skii, A., Fedorchuk, V.: General Topology I: Basic Concepts and Constructions Dimension Theory, vol. 17. Springer, Berlin (2012)
5. AtlanMod Team (Inria, Mines-Nantes, Lina). EMF random instantiator. https://github.com/atlanmod/mondo-atlzoo-benchmark/tree/master/fr.inria.atlanmod.instantiator
6. Batot, E., Sahraoui, H.A.: A generic framework for model-set selection for the unification of testing and learning MDE tasks. In: MoDELS, pp. 374–384. ACM (2016)
7. Baudry, B., Monperrus, M., Mony, C., Chauvel, F., Fleurey, F., Clarke, S.: Diversify: ecology-inspired software evolution for diversity emergence. In: CSMR-WCRE, pp. 395–398. IEEE (2014)
8. Bergmann, G., Debreceni, C., Ráth, I., Varró, D.: Query-based access control for secure collaborative modeling using bidirectional transformations. In: MoDELS, pp. 351–361. ACM (2016)
9. Bertolino, A., Daoudagh, S., Kateb, D.E., Henard, C., Traon, Y.L., Lonetti, F., Marchetti, E., Mouelhi, T., Papadakis, M.: Similarity testing for access control. Inf. Softw. Technol. **58**, 355–372 (2015)
10. Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E.: XACMUT: XACML 2.0 mutants generator. In: ICST Workshops, pp. 28–33. IEEE (2013)
11. Bordbar, B., Anastasakis, K.: UML2ALLOY: A tool for lightweight modelling of discrete event systems. In: IADIS AC, pp. 209–216. IADIS (2005)
12. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: ISSRE, pp. 85–94. IEEE (2006)

13. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: ICFEM, volume 7635 of LNCS, pp. 198–213. Springer (2012)
14. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: ASE, pp. 547–548. ACM (2007)
15. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: ICST workshops, pp. 73–80. IEEE (2008)
16. Darabos, A., Pataricza, A., Varró, D.: Towards testing the implementation of graph transformations. In: GTVMT, ENTCS. Elsevier (2006)
17. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS 2008, volume 4963 of LNCS, pp. 337–340. Springer (2008)
18. Debreceni, C., Bergmann, G., Ráth, I., Varró, D.: Enforcing fine-grained access control for secure collaborative modelling using bidirectional transformations. Softw. Syst. Model. **18**, 1737–1769 (2017)
19. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Softw. Syst. Model. **8**(4), 479–500 (2009)
20. Fleurey, F., Baudry, B., Muller, P., Le Traon, Y.: Towards dependable model transformations: qualifying input test data. Softw. Syst. Model. **8**, 185–203 (2007)
21. Godik, S., T. M. (eds): eXtensible access control markup language (XACML) version 1.0. 02 (2003)
22. Gogolla, M., Vallecillo, A., Burgueño, L., Hilken, F.: Employing classifying terms for testing model transformations. In: MoDELS, pp. 312–321. IEEE (2015)
23. Gómez, A., Mendialdua, X., Bergmann, G., Cabot, J., Debreceni, C., Garmendia, A., Kolovos, D.S., de Lara, J., Trujillo, S.: On the opportunities of scalable modeling technologies: an experience report on wind turbines control applications development. In: ECMFA, volume 10376 of LNCS, pp. 300–315. Springer (2017)
24. González, C.A., Cabot, J.: Test data generation for model transformations combining partition and constraint analysis. In: ICMT, volume 8568 of LNCS, pp. 25–41. Springer (2014)
25. Guerra, E., Soeken, M.: Specification-driven model transformation testing. Softw. Syst. Model. **14**(2), 623–644 (2015)
26. Hilken, F., Gogolla, M., Burgueño, L., Vallecillo, A.: Testing models and model transformations using classifying terms. Softw. Syst. Model. **17**, 885–912 (2018)
27. Hu, C.T., Kuhn, D.R., Yaga, D.J.: Verification and test methods for access control policies/models. Special Publication (NIST SP) 800-192, National Institute of Standards and Technology (2017)
28. Iyengar, S.S.: Metadata driven system for effecting extensible data interchange based on universal modeling language UML, meta object facility MOF and extensible markup language XML standards (2005)
29. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **2**, 256–290 (2002)
30. Jackson, E.K., Levendovszky, T., Balasubramanian, D.: Reasoning about metamodeling with formal specifications and automatic proofs. In: MoDELS, volume 6981 of LNCS, pp. 653–667. Springer (2011)
31. Jackson, E.K., Simko, G., Sztipanovits, J.: Diversely enumerating system-level architectures. In: EMSOFT, pp. 11:1–11:10. IEEE (2013)
32. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
33. Kang, E., Jackson, E., Schulte, W.: An approach for effective design space exploration. In: Monterey Workshop, volume 6662 of LNCS, pp. 33–54. Springer (2010)
34. Le Berre, D., Parrain, A.: The sat4j library, release 2.2. J. Satisf. Boolean Model. Comput. **7**, 59–64 (2010)
35. Martin, E., Xie, T.: A fault model and mutation testing of access control policies. In: WWW, pp. 667–676. ACM (2007)
36. Micskei, Z., Szatmári, Z., Oláh, J., Majzik, I.: A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In: KES-AMSTA, volume 7327 of LNCS, pp. 504–513. Springer (2012)
37. Mottu, J., Baudry, B., Traon, Y.L.: Mutation analysis testing for model transformations. In: ECMDA-FA, volume 4066 of LNCS, pp. 376–390. Springer (2006)
38. Mottu, J., Sen, S., Cadavid, J.J., Baudry, B.: Discovering model transformation pre-conditions using automatically generated test models. In: ISSRE, pp. 88–99. IEEE (2015)
39. The Object Management Group. Object Constraint Language, v2.0 (2006)
40. Ratiu, D., Voelter, M.: Automated testing of DSL implementations: experiences from building mbeddr. In: AST@ICSE, pp. 15–21. ACM (2016)
41. Reid, S.: An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In: IEEE METRICS, pp. 64–73. IEEE (1997)
42. Rensink, A.: Isomorphism checking in GROOVE. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. (2006). https://doi.org/10.14279/tuj.eceasst.1.77.84
43. Rensink, A., Distefano, D.: Abstract graph transformation. Electron. Notes Theor. Comput. Sci. **157**, 39–59 (2006)
44. Reps, T.W., Sagiv, S., Wilhelm, R.: Static program analysis via 3-valued logic. In: CAV, volume 3114 of LNCS, pp. 15–30. Springer (2004)
45. Scheidgen, M.: Generation of large random models for benchmarking. In: BigMDE@STAF, volume 1406 of CEUR Workshop Proceedings, pp. 1–10. CEUR-WS.org (2015)
46. Schönböck, J., Kappel, G., Wimmer, M., Kusel, A., Retschitzegger, W., Schwinger, W.: TETRABox—a generic white-box testing framework for model transformations. In: APSEC, pp. 75–82. IEEE (2013)
47. Semeráth, O., Barta, Á., Horváth, Á., Szatmári, Z., Varró, D.: Formal validation of domain-specific languages with derived features and well-formedness constraints. Softw. Syst. Model. **16**, 357–392 (2017)
48. Semeráth, O., Nagy, A.S., Varró, D.: A graph solver for the automated generation of consistent domain-specific models. In: ICSE, pp. 969–980. ACM (2018)
49. Semeráth, O., Varró, D.: Graph constraint evaluation over partial models by constraint rewriting. In: ICMT, volume 10374 of LNCS, pp. 138–154. Springer (2017)
50. Semeráth, O., Varró, D.: Iterative generation of diverse models for testing specifications of DSL tools. In: FASE, volume 10802 of LNCS, pp. 227–245. Springer (2018)
51. Semeráth, O., Vörös, A., Varró, D.: Iterative and incremental model generation by logic solvers. In: FASE, volume 9633 of LNCS, pp. 87–103. Springer (2016)
52. Sen, S., Baudry, B., Mottu, J.: Automatic model generation strategies for model transformation testing. In: ICMT, volume 5563 of LNCS, pp. 148–164. Springer (2009)
53. The Eclipse Project. Eclipse modeling framework. www.eclipse.org/emf
54. The Eclipse Project. EMF DiffMerge. http://wiki.eclipse.org/EMF_DiffMerge
55. The Eclipse Project. MDT Papyrus. http://www.eclipse.org/modeling/mdt/papyrus/
56. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: TACAS, volume 4424 of LNCS, pp. 632–647. Springer (2007)
57. Torrini, P., Heckel, R., Ráth, I.: Stochastic simulation of graph transformation systems. In: FASE, volume 6013 of LNCS, pp. 154–157. Springer (2010)
58. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: an integrated

development environment for live model queries. Sci. Comput. Program. **98**, 80–99 (2015)
59. Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal specification and testing of model transformations. In: SFM, volume 7320 of LNCS, pp. 399–437. Springer (2012)
60. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**, 214–234 (2007)
61. Varró, D., Semeráth, O., Szárnyas, G., Horváth, Á.: Towards the automated generation of consistent, diverse, scalable and realistic graph models. In: Graph Transformation, Specifications, and Nets, volume 10800 of LNCS, pp. 285–312. Springer (2018)
62. Wang, J., Kim, S., Carrington, D.A.: Verifying metamodel coverage of model transformations. In: ASWEC, pp. 270–282. IEEE (2006)
63. Yakindu Statechart Tools. Yakindu. http://statecharts.org/