Science of Computer Programming 215 (2022) 102745

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

User-driven diverse scenario exploration in model finders

Robert Clarisó [a,*], Jordi Cabot [a,b]

[a] Universitat Oberta de Catalunya (UOC), Spain
[b] ICREA, Spain

A R T I C L E   I N F O

Article history:
Received 22 December 2020
Received in revised form 17 September 2021
Accepted 28 October 2021
Available online 9 November 2021

Keywords:
Model-driven engineering
Verification and validation
Testing
Clustering
Diversity

A B S T R A C T

Model finders can build instances of declarative specifications that satisfy a set of correctness constraints. Some model finders ensure some degree of *diversity* among the instances they compute. Nevertheless, each model finder uses its own definition of diversity, that may or may not match designer intent.

In this paper, we propose a procedure that enables designers to capture the desired notion of diversity they are looking for. Using a simple *domain-specific language*, they can specify what elements in the specification are relevant when comparing the differences between two instances. This information can then be used to make *any* model finder diversity-aware while using it as a black box. As a proof of concept, this approach has been implemented on top of the Alloy Analyzer.

1. Introduction

The structure and behavior of a software system can be described by means of *software models*, using notations such as Alloy [1], graph-based formalisms [2] or UML/OCL [3]. These notations describe software systems at a high level of abstraction, hiding implementation details while preserving its salient features. Analyzing these models can reveal complex faults in the underlying systems.

In this analysis, the key assets for checking the correctness of software models are *model finders* [4], tools capable of computing *instances* of a model that satisfy or violate a set of constraints and properties of interest. Each model finder targets a particular modeling notation and uses a different reasoning engine, such as theorem provers [5], search-based algorithms [6,7], query containment [8], term rewriting [9], or constraint solvers, such as satisfiability (SAT) [10,11], satisfiability modulo theories (SMT) [12,13,7] or constraint programming (CP) [14]. Among other applications, the generated instances can be employed for *verification* ("is the system correct according to some criteria?"), *validation* ("does the system meet the intent of the designers?") or *testing* ("does the system behave as expected in a particular scenario?").

For verification purposes, it is usually enough to check the existence or absence of an instance which either proves or disproves the property of interest, *e.g.*, searching for a counterexample. However, for testing and validation purposes several instances are usually required to increase our confidence in the correctness of the model. It is highly desirable that those instances exhibit *diversity*, *i.e.*, distinct configurations of the system and interesting corner cases [15]. Lack of diversity may make validation and testing more time consuming, as the analysis includes almost-duplicate instances that do not provide added value; and less effective, as the sample of instances may fail to include relevant scenarios.

* Corresponding author.
E-mail address: rclariso@uoc.edu (R. Clarisó).

https://doi.org/10.1016/j.scico.2021.102745

Nevertheless, most model finders focus on efficiency and expressiveness of the input modeling notation, so few of them ensure diversity of the generated instances [16,2,15,17,18]. In these few, diversity assurance is integrated into the solver: it guides the search process to look for diverse instances. However, this integration makes it harder to transfer the proposed methods to other solvers and notations. Thus, designers are limited in terms of expressiveness (*e.g.*, no support for integer or string attributes [2,18,15] or dynamic properties [16,15,7,17]) and cannot benefit from additional features provided by others model finders (*e.g.*, computation of minimal instances [19] or support for max-satisfiability [20]).

Another challenge is how diversity is defined within model finders. Deciding whether a set of instances is sufficiently diverse depends on the domain and the specific problem being solved. For example, an attribute “password” may be irrelevant when testing properties not related to security; and when testing password strength, having different passwords of the same length may not be enough, as we may be interested in covering passwords of different length.

This paper proposes a method for *distilling diverse instances* in the model finder output. Given a set of instances computed by a model finder, this approach aims to *cluster* them into categories according to their *similarity*. Selecting a representative instance from each category ensures diversity while reducing testing and validation time, as redundant instances can be safely discarded. As an advantage, the method is (1) solver- and notation independent and (2) supports defining a custom notion of similarity. As a drawback, this method does not *force* the model finder to look for diverse instances, it only distills the most diverse ones a posteriori.

Our approach uses a graph-based representation to encode relevant information about instances, *i.e.*, their *structure* (the existing objects and the links between them), *typing* (the specific type of each object) and *attribute values*. A domain-specific language (DSL) lets designers create *diversity scripts* which specify the information that should appear in these graphs, allowing an intuitive way to customize similarity and diversity. Then, graph similarity is measured using *graph kernels* [21,22], a family of methods for computing distances among graphs.

This paper extends our previous work in [23], which introduced the use of graph kernels and clustering to improve diversity. In particular, we provide the following contributions: (1) a mechanism to customize the notion of diversity; (2) its implementation on top of the Alloy Analyzer; (3) a description of how the approach can be applied to modeling notations other than Alloy; and (4) an extended discussion of related work in this area.

Similarity versus other notions of diversity. There is a fundamental difference between targeted diversity notions (*e.g.*, guided by coverage) and the notion of similarity used in this paper.

Coverage is an *external* metric for specifying the desired diversity characteristics of the output of a model-finder. It requires an external source of information (*e.g.*, an automaton, a constraint, an imperative program, ...) from which the degree of coverage can be measured. Such metric can guide the search process of the model finder, or establish that no more instances are required.

On the other hand, similarity is an *internal* metric for diversity in the output of the model finder: it only requires the instances themselves. No other external source of information is required (other than a diversity script) and similarity is measured in terms of connectivity patterns, types and attribute values of the instances. Thus, it is useful in scenarios where the goal is unknown or incompletely specified. For instance, for validation purposes we do not know a priori what type of information will be relevant to the designer. Similarly, in the case of testing, we may not know which methods will be tested using the instances that the model finder is generating. Therefore, in these scenarios we need to compute instances that are diverse *per se* rather than relying on an externally defined goal.

Paper organization. The remainder of the paper is structured as follows. Section 2 presents an overview of the method illustrated with a simple example. Then, we describe the four steps of our method: the default *abstraction* process for transforming instances into graphs (Section 3); the definition of *custom diversity notions* and its integration in our graph-based representation (Section 4); *graph kernels* (Section 5), the framework for computing similarities among graphs; and *clustering* algorithms that can use this similarity to build groups of related instances (Section 6). Section 7 presents some experimental results of the application of this method. After that, Section 8 describes previous work on diversity and model finding. Finally, Section 9 outlines the conclusions and areas for future work.

2. Method overview

The overview of our approach for identifying diverse instances in model finder output is depicted in Fig. 1. Our **input** is a set of instances computed by a model finder. Optionally, we may also provide a definition of the type of diversity we are looking for. Then, our **output** is a set of clusters grouping those instances according to their similarity. From this output, it is possible to select a representative instance for each cluster, *e.g.*, choosing the smallest instance according to a user-defined metric.

The method can be divided into four steps:

1. **Graph encoding**: First, each instance is abstracted as a labeled graph, where labels store type and attribute value information and the underlying graph captures the objects and the links among them.
2. **Diversity refinement**: The information in the labeled graph is filtered, aggregated and abstracted to meet the diversity specification provided by the user. This is an optional step: if no specification is given, the default graph encoding is not modified.

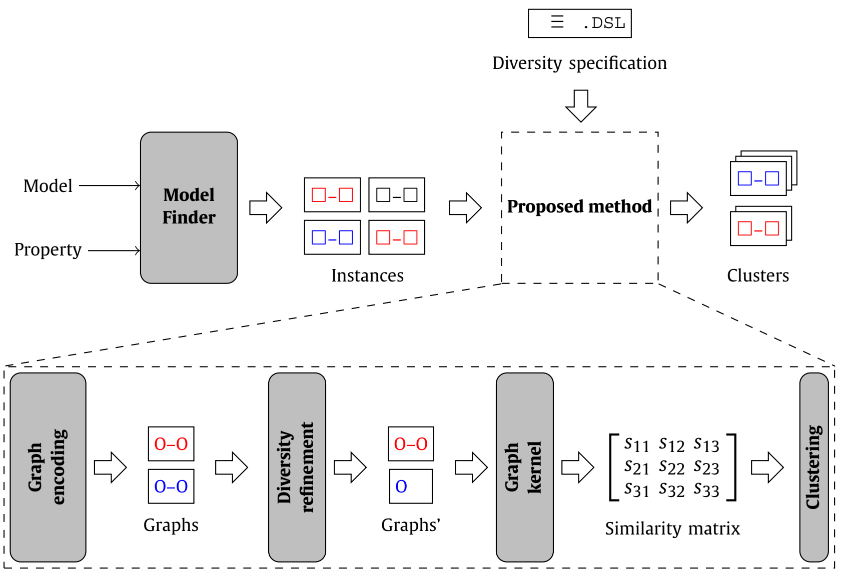


Fig. 1. Overview of the method presented in this paper.

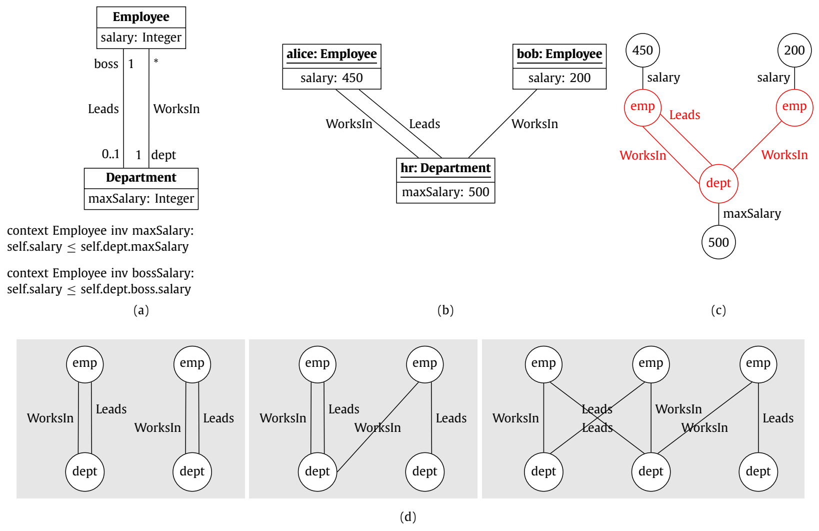


Fig. 2. Motivating example: (a) UML/OCL class diagram; (b) Sample instance; (c) Encoding of the instance as a labeled graph; (d) Graph shapes of the three clusters.

3. **Graph kernel:** Then, the pairwise similarity among the n graphs is computed using a state-of-the-art labeled graph comparison technique. The result of this computation is a $n \times n$ symmetric matrix S where each cell S_{ij} measures the similarity between graphs i and j. This notion of similarity measures repeated structures between the two graphs (*i.e.*, matching labels and connectivity patterns). Thanks to our graph encoding, which stores type and attribute value information as labels, this means that our similarity metric is implicitly considering types, values and topological information simultaneously.
4. **Clustering:** Finally, the similarity data is used by a clustering procedure to classify instances into groups of similar instances. The most suitable number of groups is determined by using *clustering validity indices*, which measure whether elements in the cluster are similar to each other and different from elements in other clusters.

To illustrate how the method works and the type of results it can achieve, we will use the UML class diagram in Fig. 2(a). This model describes the relationships between employees who work in or lead a department. There are two constraints regarding the salary, defined as OCL invariants: all salaries must be below a salary threshold and also below the salary of the department's director.

To be usable in practice, this model should be *strongly satisfiable* [14]: it should have some instance where all integrity constraints are satisfied with each class having a non-empty population. In our example, the class diagram is satisfiable and a potential solution is the instance shown in Fig. 2(b). Instances like this can then be used for validating and testing the UML/OCL model.

We have used the USE Model Validator [11] to generate 25 valid instances for this model. By manually inspecting these instances, we can easily realize that most of them are very similar, *i.e.*, isomorphic graphs with the same connectivity pattern between objects. A designer would be interested in a smaller and more diverse set of instances that gives the same or even more information as the 25 original ones. We explain next how this can be achieved with our method.

Applying our method, each object diagram is abstracted as a labeled graph. As an example, Fig. 2(c) shows the abstraction for the object diagram in Fig. 2(b). As designers, we can customize our diversity analysis by deciding to focus only on objects and their relationships, abstracting away attribute values. That is, we may decide not to consider differences in salaries or maximum salaries when comparing different instances if for our purposes their values are irrelevant. In this way, we keep only the part of the graphs that is stored in red, which still provides information about the number of employees, departments and how they are related. We then apply hierarchical clustering to our 25 graphs using the similarity information provided by a graph kernel algorithm. From the results, validity indices recommend choosing 3 clusters. Thus, we have discovered that out of the 25 instances, there are only 3 types of solutions worth considering. The common pattern in each cluster is depicted in Fig. 2(d). In particular, 21 of the 25 instances are isomorphic to the leftmost graph shape, only changing attribute values between them. Being aware of this redundancy greatly helps a modeler identify relevant instances without having to deal with almost-duplicates.

Notice that one cluster identified by our method (the middle one) highlights a potential problem in the model: a department where the director works in another department. Considering our domain knowledge, directors usually belong to the department that they lead, specially considering that according to the class diagram, each employee can only work in one department. This is a corner case worth studying:

- From a testing perspective, we need to check that the implementation considers this special case correctly, and thus we need to consider this instance as a test case.
- From a validation perspective, the designer might have included this scenario by mistake. Showing this scenario to the designers is relevant to make sure that this instance is valid or help them restrict the class diagram to avoid it altogether.

The following sections describe the different phases of our approach in more detail: Section 3 focuses on the graph abstraction that transforms the instance into a labeled graph; Section 4 describes how users can customize the notion of diversity; Section 5 discusses the *graph kernels* used to measure the similarity among pairs of graphs; and Section 6 examines how the similarity information is used to compute clusters.

3. Graph encoding

Our method for improving diversity aims to support different modeling notations and model finders, as well as taking advantage of off-the-shelf graph comparison algorithms. Thus, in this section, we describe how we translate the instances generated by the model finders for a given modeling scenario into labeled undirected graphs.

Subsection 3.1 describes the different types of instances that can be produced by existing model finders and how each of them is encoded into a set of edges and labels in a graph representing it. Among them, Alloy is the one providing more complex instances. For this reason, the other subsections focus on describing Alloy models (Subsection 3.2), Alloy instances (Subsection 3.2) and how Alloy instances are encoded into graphs (Subsection 3.4). Finally, Subsection 3.5 describes how this graph encoding can also be applied to other type of instances (*e.g.*, traces) produced by other formal method tools.

3.1. Instances of declarative models

Several modeling languages aim to provide clear and succinct descriptions of the structure of a software system. Depending on the modeling notation used to specify models and the model finder used in their analysis, the resulting instances will have a different structure.

For example, UML class diagrams annotated with invariants written in OCL (Object Constraint Language) can be analyzed using model finders such as PLEDGE [7], USE [11], UMLtoCSP [14] or AuRUS [8], among many others. The model finder would search for an instance of the class diagram that satisfies all OCL invariants and UML graphical constraints. This instance would be then returned back to the user in the form of an *object diagram*, a graphical representation describing a set of objects, each with concrete values for its attributes, and a set of links among these objects.

On the other hand, graph-based solvers such as VIATRA [24] compute labeled graphs where vertices represent objects, edges the links between them, and labels attach type information to vertices. This graph may or may not include attribute values [25].

Finally, the Alloy notation [1] allows the declarative specification of models using relational logic and their analysis using a tool (the Alloy Analyzer). Checking a property with Alloy yields a *snapshot*, a set of tuples for the different relations in our model that fulfill the property being checked (a more precise definition is provided later in this section).

Instances produced by model finders contain some common features regardless of the modeling notation or the specific model finder. First, an instance is populated by individuals called *objects* (object diagram), *atoms* (Alloy) or *vertices* (graph-based notations). Each individual has a *type* (*signature* in Alloy) and it can have *attributes* of a basic type that are given a value and *relationships* (object diagram) or *edges* (graph-based notations) to other individuals within the instance.

Intuitively, the vertices of the graph will describe the object elements in the instance, while the edges will describe the relationships among them. Labels are integer values assigned to vertices. Labels will be used to describe information such as the type of each element or the values of attributes.

The complexity of this graph encoding depends on the structure of the instances provided by the model finder. Our approach provides a specific solution for each type of output. As shown in Fig. 2, the abstraction of object diagrams is in many cases straightforward according to this pattern: objects and attributes become vertices, links become edges, and types and attribute values become labels. UML/OCL class diagrams may also include more complex features such as associative classes or n-ary associations. Nevertheless, without loss of generality, these features can be translated into classes, binary associations and additional OCL constraints [26]. Thus, it is sufficient to encode objects and links between pairs of classes.

Similarly, the mapping from instances in graph-based modeling notations is also trivial: the vertices and edges of the original graph are preserved while the type of each element is used as a label for the corresponding vertex. Nevertheless, the transformation from the relational notation used by Alloy is more involved. Thus, we devote the remainder of this Section to formalize the abstraction of Alloy instances.

3.2. Alloy models

An Alloy specification is defined as a collection of *signatures* and *constraints*, followed by a *command*.

Signatures (`sig`) describe the data in the model. Each signature has a unique name and represents a set of atoms, the base individuals in Alloy's logic. Signatures can have *fields* which take values for each atom of the signature. These values can be basic data types like integers, other signatures or complex values like functions or sets. Internally, these values are managed as *relations*, collections of tuples with the same *arity* (number of elements).

It is possible to define a hierarchy among signatures (`extends`). Moreover, fields and signatures may have *multiplicity constraints* limiting their population, *e.g.*, `one` (exactly one) or `lone` (at most one). In addition to user-defined signatures, Alloy provides some *built-in signatures* to describe common data types such as booleans, integers, strings or sequences.

Regarding constraints, there are different types: *facts* (`fact`) describe invariants that should always hold; *assertions* (`assert`) state desired properties that should be checked; and *predicates* (`pred`) are reusable constraints where some elements are passed as parameters. Each constraint can be defined using a mixture of logical operators (*e.g.*, `and`, `not` or `implies`), relational operators (*e.g.*, dot join or transpose) and quantifiers (*e.g.*, `all` or `some`).

Finally, commands instruct the solver which constraint should be analyzed and the *scope* (number of atoms) that should be considered for each signature. Command `check` searches for a counterexample of an assertion, while command `run` searches for an example of a predicate.

3.3. Alloy snapshots

Executing a command with the Alloy Analyzer may yield one of two outcomes: either no instance within the scope satisfies the constraints or an instance has been found. This search is exhaustive within the scope defined by the user, but the lack of an instance is a not proof of its absence in some larger scope. Instances are called *snapshots* in the Alloy terminology.

An Alloy snapshot is defined by the following elements:

- A list of signatures, including both built-in and user-defined signatures.
- A list of relations, each one with a fixed arity n.
- A list of *free variables* in the model, *e.g.*, parameters of predicates and existentially quantified variables.
- For each signature, a set of atoms.
- For each relation with arity n, a set of tuples of n atoms.
- For each free variable with arity n, a *witness*, *i.e.*, a set of tuples of n atoms.

That is, when checking for a property with existential quantifiers, Alloy not only answers whether it is satisfied or not: if it holds, it also provides an example of a specific value of the quantified variable (the witness) for which the property holds.

3.4. From snapshots to graphs

We need to define how to translate: (1) built-in signatures, (2) user-defined signatures and (3) relations. As witnesses are a special type of relation, we do not need to treat them separately.

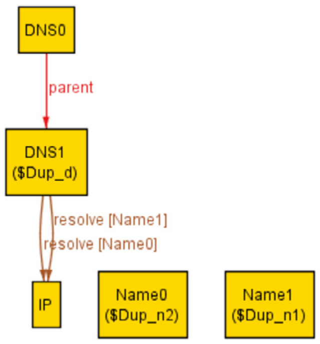
Regarding built-in signatures, we need to make sure that each value will be given the same label in different snapshots: an integer like `7` and a string like "John" should be considered equal among different snapshots. Thus, the first step is

```
sig Name, IP {}
sig DNS {
   parent: lone DNS,
   lookup: Name -> lone IP
}

pred Dup[n1, n2: Name] {
  some d: DNS | (n1 != n2) and
  (d.lookup[n1] = d.lookup[n2])
}

// Find names with same IP
run Dup for 2
```

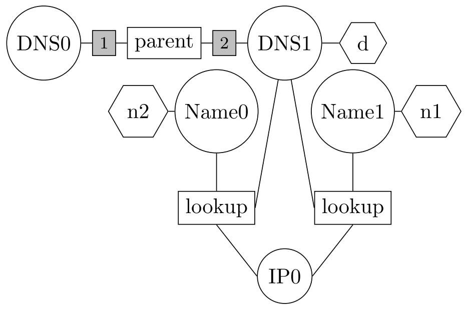
(a)

```
Atoms
Name   = {Name0, Name1}
IP     = {IP0}
DNS    = {DNS0, DNS1}

Relations
parent = {DNS0->DNS1}
lookup = {DNS1->Name0->IP0,
          DNS1->Name1->IP0}

Witnesses
Dup_n1 = {Name1}
Dup_n2 = {Name0}
Dup_d  = {DNS1}
```

(b)



(c)



(d)

Fig. 3. Example of graph abstraction: (a) Alloy model; (b) Alloy snapshot in textual format; (c) Alloy snapshot depicted graphically; (d) Abstracted graph.

traversing the set of snapshots being abstracted to construct a *vocabulary of values*. In this way, we compute a *unique label* for each value of a basic type.

1. **Built-in signatures:** We create a vertex for each atom in these signatures, plus a vertex for each built-in value (string, integer or sequence) used in the model. We label each vertex with the unique label for that built-in value.
2. **User-defined signatures:** We create a vertex for each atom. It is labeled with its signature, *i.e.*, the innermost signature in the signature hierarchy to which it belongs.
3. **Relations:** We create a vertex v for each tuple, labeled with the name of the relation. Then, for each i-th element in the tuple, we create a vertex[1] labeled with i connected to both v and the vertex of the corresponding value.

Fig. 3 shows an example of this abstraction process. The Alloy model in Fig. 3(a) describes a DNS server lookup process. We want to validate the potential scenarios in this process, for instance, whether two names may resolve to the same IP address. To do that, Alloy finds example instances, highlighting the offending names (`n1` and `n2`) and DNS (`d`). Fig. 3(b) and (c) show one sample Alloy instance in textual and graphical format.[2]

The corresponding graph abstraction is depicted in Fig. 3(d). For clarity, vertices are depicted in a different shape according to their origin: circles for atoms; rectangles for relations (white) and positions within relations (grayed); and hexagons for witnesses.

3.5. *Beyond snapshots*

While some formal method tools focus on the analysis of structural properties of a system, others are also concerned with dynamic properties. For instance, model checkers compute *traces*, *i.e.*, sequences of system states that satisfy or fail to comply with a given temporal property. Even though the Alloy Analyzer provides model checking capabilities, the instances produced by other formal method tools may differ significantly from Alloy snapshots.

[1] The intermediate vertex is omitted when the position i can be inferred: no other position in the relation has a compatible signature, *i.e.*, with a common supertype.

[2] Alloy's instance viewer depicts atom `IP0` as `IP` for brevity, given that there is only one atom in signature `IP`.

```
one sig BTrace {
  initial: one BState,
  final:   one BState
 }
sig BState {
  incoming: lone BTransition,
  outgoing: lone BTransition
}
sig BTransition {
  name: one BOperation,
  parameters: set BValue,
  returnValues: set BValue,
  src: one BState,
  tgt: one BState
}
sig BOperation {}
abstract sig BValue {}
sig BInteger { value: Int }
sig BSet     { value: set BValue }

assert { no BTrace.initial.incoming }
assert { no BTrace.final.outgoing }
assert { all s: BState | (s != BTrace.initial) implies some s.incoming }
assert { all s: BState | (s != BTrace.final) implies some s.outgoing }
assert { all t: BTransition | t.src.outgoing = t and t.tgt.incoming = t }
```

Fig. 4. Alloy specification of B traces.

In this section, we discuss a general strategy that enables designers to apply the method proposed in this paper to a large variety of formal method tools. Let us consider a target formal methods tool, which computes instances or traces in a given *output formalism*. Taking advantage of the expressiveness of the Alloy language, we propose to define an *embedding* [27] of this output formalism in the Alloy language. That is, we will define an Alloy specification (*spec*) that describes the output formalism. Then, each trace of the target tool can be translated into the corresponding snapshot of *spec*, an Alloy specification. This makes it possible to apply the graph encoding for Alloy snapshots described in subsection 3.4 to the instances of our target tool.

Traces in ProB. ProB [28] is an animator and model checker for the B-Method [29] and many other formalisms such as Z, CSP or Promela. For instance, in B specifications ProB can compute traces that lead to an error state where an invariant is not satisfied or a deadlock is reached.

Let us consider the language of ProB traces for specifications written in B [30]. A *trace* is a finite and non-empty sequence of *states* starting from an *initial state*. Between each pair of states there is a *transition* representing the execution of an operation. Each transition is labeled with: the *name* of the operation being invoked; the (possibly empty) finite list of *parameter values* passed to the operation; and the (possibly empty) finite list of *return values*.

For the sake of simplicity, we will restrict the types of values that we consider to integers (INT) and sets. In this scenario, the Alloy specification in Fig. 4 can be used to describe B traces. The proposed translation is a *deep embedding*, given that each element in the B trace language is modeled explicitly using Alloy constructs. A much simpler *shallow embedding* would also be possible.

Using an embedding such as this one, a formal method tool can translate its output instances into an Alloy snapshot (a text file). Then, our method can be applied directly, encoding the Alloy snapshot into a graph.

To conclude, we do not claim that this deep embedding process produces optimal graph encodings. It is possible to define more compact encodings for a particular tool. This section only intends to illustrate the degree of generality of the proposed approach.

4. Customizing diversity

The graph encoding defined in Section 3 transforms each instance into a graph that captures its type, value and topological information. Nevertheless, this mapping assumes that all elements in the specification are relevant from the point of view of diversity. As we have seen, this is not always desirable, given the different ways in which the instances produced by the model finder can be used.

Some approaches aimed at achieving diversity use *uniform sampling* [31–34] as their goal: achieving a uniform distribution among solutions. Nevertheless, the desired notion of diversity may be more complex (a target probability distribution, a partition into meaningful classes), and specific to a domain or even a particular problem [7,17]. The most precise way to control diversity would be to allow designers to define the *distance* function to be used when comparing two instances. Nevertheless, writing an ad hoc distance function is not a trivial task, as it needs to compute and combine information about types, attribute values and the structure of the instance.

In this Section, we propose a domain-specific language that designers can use to select and refine the model features that are important for them when determining the level of diversity. Using this DSL, designers can specify the submodel of interest or aggregate some properties of the elements in the model. Any information from the instance outside this submodel is still generated by the model finder, but it is filtered out when it comes to measuring and enforcing diversity among instances.

This DSL aims to be concise, usable and expressive. It offers high-level concepts that will be familiar to designers rather than implementation details such as the distance function being used. The language comprises a set of modeling primitives in order to:

1. Define the signatures, relations and attributes that are relevant from a diversity perspective, *i.e.*, all others can be ignored.
2. Identify the fields whose values can be abstracted, *i.e.*, it is not necessary to consider their specific individual value. Instead, we would like that value to satisfy a diversity of properties of interest or to fall within different categories according to some criteria.
3. Specify the information in a relation that is only relevant in an aggregate form, *e.g.*, count the number of objects participating in the relation or compute the sum/min/max of a numeric attribute.

4.1. Relevant elements

To reduce verbosity, the DSL provides two ways of defining the submodel of interest: `default relevant` (unless otherwise noted, everything is relevant) and `default irrelevant` (unless otherwise noted, everything can be hidden). It is also possible to highlight that all model elements of a certain type (*e.g.*, witnesses or fields) are always considered relevant, *e.g.*, `default witness irrelevant`. If the designer does not make an explicit choice, we will assume relevance by default.

After defining the default behavior, the designer only needs to list the elements that deviate from this default behavior. For instance, if the default is `irrelevant` we could say `relevant <ListOfElements>`, where the `<ListOfElements>` is a comma-separated list of signatures, fields or witnesses (specified as `sig-name.*` for all fields of a signature or using `sig-name.field-name` for a specific field). Note that it is possible for a signature to be relevant while its attributes are irrelevant, but the opposite is not possible: attributes in an irrelevant signature are also irrelevant.

Example 1. Going back to our example on employees and departments from Fig. 1, a diversity script stating that attribute maxSalary in class Department can be defined as irrelevant would be:

```
default relevant
irrelevant Department.maxSalary
```

If we wanted to abstract the two attributes in our example (salary and maxSalary), we could do it as follows:

```
default relevant
irrelevant Department.maxSalary, Employee.salary
```

Example 2. Assuming we are only interested in diversity among workers and their salaries, we can specify that as:

```
default relevant
irrelevant Department
```

4.2. Abstracting values

We denote that the concrete value of a field will be replaced by a more abstract value using the following syntax:

```
abstract <Field> with <Function>
```

where `<Field>` is defined using the `sig-name.field-name` syntax and `<Function>` is the name of a function that abstracts the value of a parameter as an integer value. We consider the following types of abstractions:

- Boolean properties about the value of the field that return 0 (false) or 1 (true), *e.g.*, the predicate `isZero()` that checks whether a numeric field is equal to zero.
- Functions computing a metric or assigning a discrete category to the value of a field. For instance, `sign()`, that returns -1, 0 or 1 according to the sign of a numeric field or `length()`, that computes the length of a String field.

In addition to predefined functions like the previously mentioned ones, we support invoking user-defined functions from the Alloy specification. These functions should have a single argument (a value of this field) and produce an Int value as a result. This feature allows a lot of flexibility to this DSL by allowing the definition of custom problem-specific abstractions.

```
script         ::= /* empty */ | statement script
statement      ::= default_stmt | relevance_stmt
                 | abstract_stmt | aggregate_stmt
default_stmt   ::= DEFAULT feature relevance
feature        ::= /* empty */ | ATTRIBUTE | WITNESS | RELATION
relevance      ::= RELEVANT | IRRELEVANT
relevance_stmt ::= relevance name_list
name_list      ::= name | name , name_list
name           ::= IDENT | IDENT . * | IDENT . IDENT
abstract_stmt  ::= ABSTRACT IDENT . IDENT WITH IDENT ( )
aggregate_stmt ::= AGGREGATE IDENT . IDENT WITH IDENT ( IDENT )
```

Fig. 5. Syntax of the proposed DSL.

Example 3. Consider that for diversity purposes we want to select instances that distinguish regular companies from non-profit organizations where some workers do not receive any compensation. To target this notion of diversity, we can abstract salary in this way:

```
abstract Employee.salary with isZero()
```

4.3. Aggregating relations

Finally, we denote that a relationship can be abstracted by considering an aggregation of data from the participants in the relation in the following way:

```
aggregate <Relation> with <Operation>(<Argument?>)
```

where `<Relation>` is the name of the field where we are performing the aggregation; `<Operation>` can be either `count`, `min`, `max` or `sum`; and the optional `<Argument>` is the name of the target field whose value will be aggregated (for `min`/`max`/`sum` operations), which is not required for the `count` operation (where we only count the population).

Example 4. A diversity script that abstracts employees' details replacing them with the number of employees in each department would be the following:

```
aggregate Employee.WorksIn with count()
irrelevant Employee, Employee.*
```

Example 5. Consider that the user is looking for diversity in the total payroll of departments. The following script could be used:

```
default irrelevant
relevant Department
aggregate Department.WorksIn with sum(salary)
```

A similar script can be used to consider only the salary of the director of a department:

```
default irrelevant
relevant Department
aggregate Department.Leads with sum(salary)
```

4.4. Implementation of the DSL

The DSL is implemented as a parser written in JFlex (lexer) and CUP (syntax and semantics). The parser reads the diversity specification file and stores the settings: which signatures and fields are relevant, which fields are abstracted and with which predicate, which relationships are aggregated, ...Later, when instances are encoded as labeled graphs, these settings are used to control how the graph encoding is generated.

Fig. 5 describes the complete syntax of the DSL using the BNF notation. The identifiers appearing in the grammar can refer to either signature names (before the dot symbol), field names (after the dot symbol) or operation names (before opening parenthesis).

Meanwhile, the pseudocode in Algorithms 1 and 2 describes the semantics of the DSL by showing how it governs the graph encoding process. For the sake of clarity, we have hidden some parts of the graph encoding, such as the encoding of positions in relations, the encoding of witnesses or the encoding of aggregations. Looking more closely at the pseudocode in Algorithm 2, relevance dictates which elements are encoded in the graph: in lines 10 and 21, irrelevant signatures are not included in the graph; in line 22, irrelevant fields are also removed. Meanwhile, lines 25–26 take care of the abstraction of field values: if the designer has defined any operation for abstracting a field, the operation is applied to the value of the field and the resulting value is used instead.

```
Function EncodeSnapshotSet ($dsl$, $AS$)
    input : $dsl$: A DSL file; $AS$: a set of Alloy snapshots
    output: A set of labeled undirected graphs, one per snapshot in $S$
    // Parse diversity script
    // The settings record which signatures and fields are relevant, which fields
    // should be abstracted and which relationship should be aggregated
    settings ← parse ($dsl$);
    // Build a dictionary that maps entities to unique integer labels
    // Values shared between two or more snapshots will be given the same label
    dict ← $\emptyset$;
    foreach snapshot $as \in AS$ do
        foreach constant atom $c \in as$ do
            // If $c \in$ dict, return previously assigned integer label
            // If $c \notin$ dict, add $c$ to the dictionary and assign new label
            dict.findOrAdd ($c$);
    // Encode each snapshot separately
    $\mathcal{G} \leftarrow \emptyset$;
    foreach snapshot $as \in AS$ do
        $\mathcal{G} \leftarrow \mathcal{G} \cup$ EncodeSnapshot ($s$, settings, dict));
    return $\mathcal{G}$;
```

Algorithm 1: Preparation of the graph encoding process.

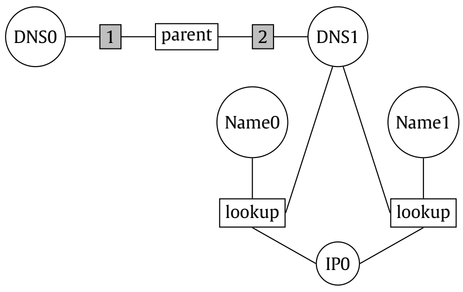
```
Function EncodeSnapshot ($as$, settings, dict)
    input : $as$: an Alloy snapshot; settings: diversity settings from the DSL; dict: label map
    output: A labeled undirected graph encoding the types, values and topology of $as$
    // Start with an empty undirected graph $G = (V, E)$
    $G \leftarrow (\emptyset, \emptyset)$;
    // Encode constants
    foreach constant atom $c \in as$ do
        label ← dict.findOrAdd ($c$) ;                              // Retrieve label for constant $c$
        $V(G) \leftarrow V(G) \cup \{$ newVertex (label, $c$) $\}$;          // Add vertex for constant $c$
    // Encode atoms in signatures
    foreach signature $s \in as$ do
        if settings.isRelevant ($s$) then                           // DSL: signature relevance
            label ← dict.findOrAdd ($s$);                           // Retrieve label for signature $s$
            foreach atom $a \in s$ do
                $V(G) \leftarrow V(G) \cup \{$newVertex (label, $a$) $\}$ ;     // Add vertex for atom $a$
    // Encode relations
    foreach relation $r \in as$ do
        label ← dict.findOrAdd ($r$) ;                              // Retrieve label for relation $r$
        foreach tuple $t \in r$ do
            $v_1 \leftarrow$ newVertex (label,$t$) ;                          // Add vertex for tuple $t$
            $V(G) \leftarrow V(G) \cup \{v_1\}$;
            foreach field $f \in t$ do
                if settings.isRelevant (signature ($f$) )
                and settings.isRelevant ($f$) then                  // DSL: field relevance
                    $a \leftarrow t[f]$;                            // Atom $a$ in field $f$ of tuple $t$
                    if $a$ is a constant atom then
                        op ← settings.abstractFieldOp ($f$);        // DSL: field abstraction
                        value ← op (a);                             // Apply field abstraction
                                                                    // operation (if any)
                        $v_2 \leftarrow$ findVertex (value, $V(G)$);            // Recover vertex for value
                        if not found then                           // New constant value
                            label ← dict.findOrAdd (value);         // Assign new label
                            $v_2 \leftarrow$ newVertex (label, value);         // Add new vertex
                            $V(G) \leftarrow V(G) \cup \{v_2\}$
                    else
                        $v_2 \leftarrow$ findVertex ($a$, $V(G)$);               // Recover vertex for atom $a$
                    $E(G) \leftarrow E(G) \cup (v_1, v_2)$;
    return $G$;
```

Algorithm 2: Graph encoding for a single Alloy instance.

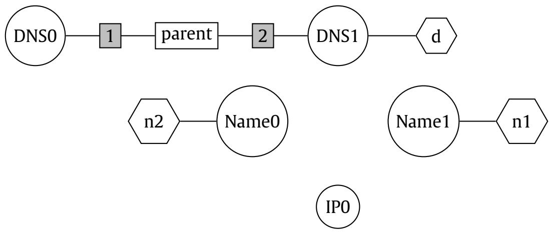
4.5. Discussion

In order to illustrate the capabilities of this DSL, we revisit the example specification on DNS server lookup presented in Fig. 3. Starting from the instance from Fig. 3(d), we define three diversity scripts and present their impact in Fig. 6.

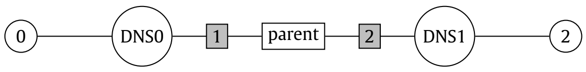
The first script (top of Fig. 6) removes the witnesses from the graph encoding, in case the user is not interested in this information. The second (middle of Fig. 6) removes the `lookup` relation, keeping the remaining information about the signatures and atoms that participated in the relation. Finally, the last script (bottom of Fig. 6) only considers the information from the model that concerns the DNS server. To this end, it hides most information within the instance but



Ignore diversity among witnesses
```
  default irrelevant witness
```



Ignore diversity in `lookup`
```
  irrelevant DNS.lookup
```



Look for diversity in DNS (parent, number of lookups)
```
  default irrelevant
  relevant DNS, DNS.parent
  aggregate DNS.lookup with count()
```

Fig. 6. Sample customizations of the graph encoding from Fig. 3.

also adds new information to it: the number of name-to-IP lookups in which the server is involved, a metric which was previously implicit in the underlying structure of the graph.

The application of the diversity scripts enables the designer to define different views of the same instances. Using these different views, it is possible to consider two instances similar from one point of view but dissimilar from another. For instance, considering the bottom script, this diversity notion prioritizes having diverse hierarchies of DNS servers, while the number of IP addresses will not be taken into account when comparing two instances. Using this view focused on the DNS servers, our clustering would differ significantly[3] from another one that, for instance, concentrated on the IP addresses and names. Clearly, there is no "best" diversity notion: which view works best will depend on the domain, the problem and the goals of the designer that is generating the instances.

Section 5 will discuss how we can use these graphs to compute the distance between instances, so that we can later classify them into groups according to their similarity.

5. Graph kernels

There are different ways to compare a pair of graphs and establish the degree of similarity between them. For instance, the *edit distance* measures the number of atomic changes required to transform one graph into the other. An alternative is

[3] There are some corner cases where the clustering would remain unchanged. For instance, if all instances are equal, the proposed clustering will be the same (group all instances into a single cluster) regardless of the diversity script.

Table 1
Examples of graph kernels.

Method	Structure	Efficiency
Random walks	Walks (allows repetitions)	$O(n^3) - O(n^6)$
Shortest paths	Paths (no repetitions)	$O(n^4)$
Graphlets	Subgraphs of size k	$O(n^k)$
Weisfeiler-Lehman	Subtrees	$O(hm)$

checking for *isomorphism*[4] between the whole graphs or their subgraphs. However, these approaches have a high computational complexity and may be unsuitable for comparing large graphs or sizable collections of graphs.

An alternative approach is taken by graph kernels [21,22], a family of methods for measuring the (dis)similarity among pairs of graphs. Rather than computing an exact measure for similarity, kernels aim to provide an efficient approximation that can be computed efficiently but still captures relevant topological information about the graphs. A typical approach is counting the number of matching substructures within the graphs, like paths, subtrees or subgraphs.

Graph kernel catalog. Table 1 presents some typical kernels describing the structures they use and their complexity in terms of the number of vertices (n) and edges (m), as well as parameters of the analysis that can be tuned to select the proper trade-off between precision and efficiency (k and h).

The *random walk* kernel counts the number of walks (sequences of adjacent vertices) that match (*i.e.*, exhibit the same sequence of labels) between two graphs.

The *shortest paths* kernel compares the list of shortest paths between two graphs, measuring similarity as the number of matches. Two paths match if they have the same distance between them and the same labels in the initial and final vertices.

The *graphlet* kernel [35] considers all possible graphs with k vertices (graphlets) and counts how many times each graphlet appears as a subgraph in the graphs being compared. Similar graphs will tend to have similar counts of each type of subgraph. The choice of k will provide a trade-off between a more precise measure of similarity (for larger values of k) and efficiency (for smaller values of k). In order to keep the number of graphlets manageable, the value used for k is usually small (*e.g.*, between 3 and 5).

Obviously, the use of more complex graph substructures enables a more precise classification of graph connectivity patterns. Nevertheless, improving precision also increases the computational complexity of the kernel. The choice of the most suitable kernel for a particular domain is an open problem [35], with no theoretical framework to provide an objective selection criteria.

The Weisfeiler-Lehman kernel. In this work, we have used the Weisfeiler-Lehman kernel [36], as it is widely considered to be the state-of-the-art graph kernel [37] and has been shown to provide good precision with an efficient computation in a variety of domains [36,38] including linked data and software analysis [39,40].

Algorithm 3 describes the Weisfeiler-Lehman (WL) kernel. The procedure computes the distance between a pair of graphs G_1 and G_2 by counting the number of common subtrees up to height h. To avoid enumerating subtrees explicitly, a characteristic label is computed for each subtree. This label is constructed iteratively: each iteration i computes the label for the tree of height i rooted in each node v ($\texttt{label}(i, v)$). Iteration 0 (line 11) uses the original labels in the graph. Then, each iteration i (lines 14-21) assigns a label to each vertex v by combining the labels of v and its adjacent vertices in iteration $i-1$. Finally, the distance between the pair of graphs is computed by counting the original labels (line 12) and the labels for subtrees up to height h (line 22) and comparing their frequencies (lines 4-6). The complexity of this procedure is $O(hm)$, with m being the number of edges in the graphs [36]. The parameter h allows us to control the trade-off between performance and precision.

Notice that thanks to how our graph abstraction process is defined (types and attribute values as labels), the similarity value computed by the kernel is implicitly taking advantage of topological, type and attribute value information from the instance.

6. Clustering

Clustering is one of the fundamental tasks in the field of Machine Learning (ML). Intuitively, it consists in the analysis of a collection of elements to identify groups of similar individuals, for a given definition of “similarity”.

6.1. Algorithm selection

Several algorithms have been proposed for this task [41]. There is no single “best” clustering algorithm: the most suitable one depends on the collection being analyzed. This is because the strategies for finding clusters can be very different. For example, *means* and *medoids* are different definitions of the “center” of a cluster, and algorithms like K-*means* and K-*medoids*

[4] Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called isomorphic if there is a mapping $f : V_1 \to V_2$ such that $\forall x, y \in V_1 : (x, y) \in E_1$ iff $(f(x), f(y)) \in E_2$.

```
Function WLKernel($G_1, G_2, h$)                                    // Weisfeiler-Lehman graph kernel
    input : $G_1, G_2$: a pair of labeled undirected graphs; $h$: an integer (the tree height)
    output: A distance measure between $G_1$ and $G_2$
    freq1 ← WLTest($G_1, h$);                                       // frequency of each label in $G_1$
    freq2 ← WLTest($G_2, h$);                                       // frequency of each label in $G_2$
    distance ← 0;                                                   // distance = difference among frequencies
    foreach label lab do
        distance ← distance + |freq1[lab] − freq2[lab]|;
    return distance;

Function WLTest($G, h$)                                             // Weisfeiler-Lehman isomorphism test
    input : $G$: a labeled undirected graph $G = (V, E)$; $h$: an integer (the tree height)
    output: A map counting the frequency of labels in $G$
    // Initially all labels x have frequency[x] = 0
    foreach vertex $v \in V(G)$ do
        label(0,v) ← label of $v$ in $G$;
        frequency[label(0,v)] ← frequency[label(0,v)] + 1
    for $i \leftarrow 1$ to $h$ do
        foreach vertex $v \in V(G)$ do
            adjacentLabels ← labels(i-1, neighbours(v, G));
            // signature = my label + sorted labels of adjacent vertices
            signature ← append(label(i-1,v), sort(adjacentLabels));
            // Assign an integer label that summarizes signature
            // Two equal signature should always receive the same label
            // Compressed labels not reused in the next iterations
            label(i, v) ← compressLabels(signature) ;
            frequency[label(i,v)] ← frequency[label(i,v)] + 1
    return frequency;
```

Algorithm 3: Pseudocode for the Weisfeiler-Lehman graph kernel [36].

aim to find the best location for those centers. On the other hand, methods like *hierarchical clustering* initially consider each element as a cluster and then iteratively merge the two nearest clusters.

In order to select which clustering algorithm should be used, the required input information should be considered:

- **Feature versus Kernel methods**: Some algorithms like K-means require each element to be described by a vector of features (relevant characteristics) of a fixed length. Meanwhile, other techniques like hierarchical clustering only require a distance (or similarity) measure among pairs of elements.
- **Target number of clusters**: Algorithms like K-means or K-medoids require knowing the target number of clusters a priori. Conversely, algorithms like hierarchical clustering do not require this information beforehand.

In our context, the elements we are trying to cluster are labeled graphs[5] abstracting the outputs of a model finder. The number of target clusters is unknown a priori. Moreover, the graphs to be clustered are not described as a feature vector: instead, the similarity metric provided by the Weisfeiler-Lehman graph kernel is used to compare them. The lack of a feature motivates us to select a hierarchical clustering method.

Hierarchical clustering [43] is a family of clustering techniques that build a hierarchy, with each element in a different cluster at the bottom and a single cluster at the top. This hierarchy can be built using a *bottom-up* or *agglomerative* strategy (by merging the two most similar clusters in the current level) or a *top-down* or *divisive* strategy (by splitting the elements in a cluster into two disjoint groups).

Agglomerative algorithms differ in the strategy used to decide which two clusters should be merged next, that is, how we decide which two clusters are the most similar. The two most popular methods are *complete linkage* and *average linkage*, which define the distance between two clusters c_1 and c_2 as the highest / average distance between elements of c_1 and c_2. Both of these approaches provides more compact clusterings than using *single linkage* (minimum distance between elements) and are the technique of choice in most works. After checking that both approaches yielded comparable results, we settled with the *average linkage* approach.

6.2. Choice of number of clusters

The output of hierarchical clustering is a *dendrogram*, a tree that describes the order in which clusters should be merged according to their similarity. A clustering is obtained when we decide where (in which level of the tree) the merging should stop. In order to decide that, we can use *cluster validity indices*, metrics that measure the quality of a clustering. In a good clustering, elements within a cluster should be very similar and very dissimilar to elements in other clusters. The metric is evaluated in each level of the tree and the clustering providing the optimal value is selected.

[5] In the ML literature, the term *graph clustering* is used with two different meanings [42]: finding densely connected subgraphs within a large graph; and grouping a collection of graphs according to their similarity. In this work we consider the latter.

Table 2
Summary of the models verified using the Alloy analyzer.

Model	Domain	Sig	Field	Fact	Pred
chord-bug-model	Distributed hashtable lookup protocol	4	8	3	15
file-system	Generic file system	7	4	0	3
firewire	Leader election (Firewire protocol)	15	16	2	15
flip-flop	Flip-flop state machine	6	8	1	2
genealogy	Genealogical relationships	5	2	4	1
grandpa	"I am my own grandfather" puzzle	3	3	3	2
philosophers	Dining philosophers problem	3	5	1	2
railway	Train safety in a railway system	4	5	3	6
reset-flip-flop	Evolution of a flip-flop	7	8	1	2

In this work, we have used the *silhouette coefficient* [44], a classical metric that measures the average distance to elements in the same cluster compared to the minimum of the average distances to elements in other clusters. It provides a value in the $[-1, 1]$ range (higher is better), where values below 0.5 signal a bad fit in the clustering. As mentioned previously, the clustering achieving the highest average silhouette width is selected as our output.

7. Experimental results

In order to assess the computational effort of the proposed method and the usefulness of its output, we have performed several experiments. These experiments aim to answer the following research questions:

RQ1. How does the execution time of the method compare to model finding?
RQ2. Do the resulting clusters provide a concise yet diverse summary of the model finder output?

Experiment design. We have analyzed a collection of Alloy models provided in the Alloy GitHub model repository.[6] Among them, we have chosen examples dealing with the generation of examples or counterexamples, rather than proving their absence. These types of models could be used for validation and testing, and thus they are the target of the proposed method. For these models, we have used the Alloy Analyzer to generate up to 100 instances (less if there are not enough valid instances available). Table 2 provides information about the size and complexity of these models: the number of signatures (**Sig**), fields (**Fields**), facts (**Fact**) and predicates (**Pred**) in each Alloy model.

Implementation. We have implemented our method as two separate components. First, we have developed a Java program that calls the latest version of the Alloy API (5.0.0) to compute a collection of instances and generate their graph encoding. This program also receives as input a DSL script (parsed using JFlex/CUP) in order to adapt the graph encoding to the desired notion of diversity. The output of this tool is stored as a set of files in GML format. Then, a R script reads the GML files, computes the graph kernel and performs the clustering. This script takes advantage of existing libraries for representing graphs (the `igraph` package[7]), similarity analysis among graphs (the `graphkernels` package[8] [45]) and clustering (the `cluster` package[9]).

The experiments have been performed on a quad-core Intel i5-760 2.8 GHz with 4 GB of RAM. On the software side, we have used Java 9.0.4 64 bits and R 3.50 64 bits. With respect to the settings, Alloy has used MiniSat as the SAT solver back-end with the highest amount of symmetry breaking (symmetry = 20). Regarding the graph kernel, the Weisfeiler-Lehman graph kernel has been used with the default number of iterations ($h = 5$).

Execution times have been measured in each step of the computation: the Alloy analysis, the graph abstraction phase and the kernel and clustering phases.

Results and discussion. Table 3 shows, for each experiment, the scope used in the analysis (**Scope**) and the number of computed instances (**Inst**). Notice that for two models there were less than 100 satisfying instances. Then, we describe the time (in milliseconds) required by Alloy to compute the instances (**Model finding**), compared to the time taken by the different steps of our method: graph abstraction (**Abst**), graph kernel (**Kern**) and clustering (**Clust**). The total time for the three steps is reported as well. Then, Table 4 lists the optimal number of clusters (**# Cl**) identified by our method and the silhouette coefficient (**Sil**). As mentioned in Section 6, the silhouette is a value in the $[-1, +1]$ range that estimates the quality of the clustering (higher is better).

Considering these results, regarding RQ1 (efficiency) the execution time of the method is always below 0.5 seconds and less than the time required by Alloy to compute the instances. This was somewhat expected, as the computational effort of our approach depends on the number of instances and their size, but it is unaffected by the hardness of finding

6 https://github.com/AlloyTools/models.
7 https://igraph.org.
8 https://cran.r-project.org/package=graphkernels.
9 https://cran.r-project.org/package=cluster.

Table 3
Experimental results: efficiency.

Model	Analysis		Model finding	Execution time			
	Scope	Inst		Abst	Kern	Clust	Total
chord-bug-model	2	52	498 ms	169 ms	90 ms	30 ms	289 ms
file-system	5	100	825 ms	165 ms	180 ms	30 ms	375 ms
firewire	2-7	100	1474 ms	209 ms	180 ms	40 ms	429 ms
flip-flop	10	100	652 ms	203 ms	180 ms	50 ms	433 ms
genealogy	6	100	830 ms	129 ms	140 ms	50 ms	319 ms
grandpa	4	48	554 ms	88 ms	70 ms	40 ms	198 ms
life	3-6	100	1681 ms	283 ms	180 ms	40 ms	503 ms
philosophers	4	100	1539 ms	157 ms	160 ms	40 ms	357 ms
railway	1-4	100	735 ms	179 ms	170 ms	30 ms	379 ms
reset-flip-flop	10	100	672 ms	250 ms	160 ms	40 ms	450 ms

Table 4
Experimental results: clustering and diversity customization.

Model	Default		Abstraction		Aggregation	
	# Cl	Sil	# Cl	Sil	# Cl	Sil
chord-bug-model	5	0.31	6	0.75	5	0.28
file-system	3	0.99	3	0.99	3	0.99
firewire	3	0.76	3	0.81	3	0.79
flip-flop	2	0.04	55	0.07	–	–
genealogy	33	0.45	32	0.72	–	–
grandpa	2	0.96	2	0.96	2	0.96
life	14	0.30	14	0.30	2	0.67
philosophers	2	0.30	10	0.37	–	–
railway	50	0.46	50	0.46	2	0.24
reset-flip-flop	14	0.48	14	0.48	–	–

Diversity notions: **Default:** Default graph encoding, **Abstraction:** Abstracting selected fields and signatures, **Aggregation:** Aggregating selected relations using `count` (population).

instances, the decisive factor in Alloy's execution time. Therefore, we can conclude that using our approach does not incur in a significant overhead with respect to using the model finder.

With respect to the scalability of our approach, let us consider the computational complexity of our method. We consider two parameters in this analysis: n, the number of instances that will be computed by the model finder; and m, the size (number of atoms, tuples in the relation and witnesses) of an instance. Graph abstraction performs a traversal of the instance, requiring $O(m)$ time. The graph kernel takes $O(m)$ time for each comparison and performs $O(n^2)$ comparisons, so in total it requires $O(m \cdot n^2)$. Finally, clustering requires $O(n^3)$ time, so the overall complexity is $O(m \cdot n^2 + n^3)$. In terms of space complexity, we require $O(m \cdot n)$ to store the n graphs, $O(n^2)$ to store the similarity matrix and perform clustering, that is, $O(m \cdot n + n^2)$ in total.

Regarding RQ2 (quality of the output), in Table 4 we have considered three different scenarios: one where we use the default encoding; one where we abstract some of the fields and signatures of the model according to the characteristics of the problem; and one where we aggregate selected relations using the `count` operation, *i.e.*, instead of keeping the specific tuples in the relation, we record the number of tuples associated with each atom.[10] We can see that the proposed number of clusters varies significantly from one model to another, and so does the silhouette coefficient:

- Models with a high silhouette (*e.g.*, file-system and grandpa) exhibit some sort of symmetry that is not being detected by the Alloy Analyzer. For instance, in file-system there is a symmetry between directory names, so in practice, it is as if Alloy was only returning the same 3 effective instances all the time. Models like this one are the scenarios where our approach is most effective.
- Models with a low number of clusters and a low silhouette (*e.g.*, flip-flop) highlight scenarios where all instances are very similar. For instance, in flip-flop the instance models 10 steps of a trace in the evolution of a flip-flop. All these traces are very similar, so no salient features can be used to classify them. Diversity can only be slightly improved for these scenarios.
- Models with a high number of clusters (*e.g.*, genealogy or railway) describe scenarios where the instances produced by the solver are already very dissimilar among them. In this case, the output of the solver was already diverse before applying our method.
- The rest of models, with an average silhouette between (0.4-0.7) illustrate a middle ground: some instances share similarities but the boundaries between each group may overlap or be hard to establish. Choosing a representative from

[10] Specifications where a `count` aggregation does not make sense, either because the specification involves a single relation or only basic signatures and fields with multiplicity one, are marked with a dash "–".

Table 5
Summary of related work considering diversity in model finding.

User-defined goal	**Redundancy avoidance**
▲ Increases and enforces diversity ▼ Requires domain knowledge	▲ Increases and enforces diversity ▼ Solver-specific
• Partial instance [10,2,19] • Target instance [46] or model [47] • Classifying terms [48] • Properties [12], patterns or metrics [49] of interest • Probability distribution [50,16] • Realistic traits [51,24]	• Minimality [19,52] • Symmetry breaking [10,14] • Distance measures [53,52,54] • Graph shape analysis [55,2] • Randomness [53,52,54] • Randomized partitioning [15]
Clustering of instances	**Clustering of meta-models**
▼ Notation-specific (object diagrams) ▼ Does not consider structure, types and attribute values simultaneously	▼ Used to compare class diagrams (not applicable to instances)
• COMODI [56,17]	• Feature vector & K-means [57] • n-grams & Hierarch. cluster. [58,59]

each cluster ensures diversity, but there is the risk (higher for lower silhouette values) of discarding relevant instances. To reduce this risk, it would be possible to select a higher number of representatives per cluster.

With respect to the customization of diversity, choosing a suitable notion of diversity may introduce changes in the proposed clustering. The changes affect both the choice of clusters and the number of clusters to be used. In some examples like `genealogy` and `chord-bug-model`, abstracting fields results in a clustering that provides a better fit for the available instances, while in others it does not improve with respect to the default one. With respect to aggregating relations, there are significant changes in some examples like `life` and `railway`. In these examples, instead of keeping track of the elements that belong to set of live states and closed gates, respectively, after the aggregation we only record the *number* of live states and closed gates. In this way, this aggregation removes details that may be unnecessary (depending on our goals) and helped us identify more concise clustering candidates for these two examples. Obviously, deciding how diversity should be customized for a particular specification will depend on the goals of user. Nevertheless, these examples show that diversity scripts have an impact on the output clustering and, thus, they can help the designer guide the choice of clusters and representative instances.

To sum up, our method can reduce the number of instances to consider while preserving diversity. Furthermore, this method provides an estimate of the quality of its result that helps designers deciding when and how to employ it.

8. Related work

Several works have considered how to improve the diversity in the output of model finders, *e.g.*, [16,2,15,17,18,48]. As shown in Table 5, we classify them into four groups: (i) methods for guiding model finding and customizing diversity; (ii) methods for redundancy avoidance in model finding; (iii) clustering methods for classifying instances; and (iv) other types of clustering applied to declarative specifications.

8.1. User-defined goal

Some works require the user to guide the model finder by setting a goal, *i.e.*, a description of what type of solution is expected. The definition of this goal can take several forms:

- The solution should include a *partial instance*. This partial instance should be completed by adding (not removing) elements [10] and it may include unknown elements which may belong to the solution or not [2].
- The solution should be as close as possible to a *target instance* [46] or any instance of a *target model* [47].
- Different solutions should cover different *partitions of values* defined by the designer using predicates called *classifying terms* [48]. For instance, for a given class the designer may only be interested in how many objects exist for that class. That partition could also be defined as: satisfying or failing to satisfy a *property* of interest [12]; the existence of relevant patterns within the instance [49]; or the value of certain metrics [49].
- The values and shape of the solution should follow a *probability distribution* defined by the designer [50,16]. In particular, some methods aim to find solutions that are *realistic*. That is, the solution should exhibit traits, properties and values for relevant metrics that are similar to those of an existing dataset of real instances [51,24].

By contrast, other methods such as [15,18] do not require any input from the designer: diversity is defined implicitly by ensuring non-equivalence or enforcing some distance metric between the output instances. However, this notion of diversity is predefined and static so it cannot be customized for particular problems.

Our solution proposes the use of a DSL to identify the *submodel of interest* that should be considered in terms of diversity. The language is intended to be simple, far from more complex *graph query* [60] or *graph transformation* [61,62] languages that allow the definition of graph patterns and navigational expressions. Nevertheless it is sufficient to take into account both the type information, attributes values and the structure of the instance. We believe it is sufficient to characterize relevant diversity notions without requiring (a) significant designer effort to craft complex patterns or expressions of interest; (b) deep knowledge about the domain or the problem; or (c) the availability of a dataset of realistic instances. Moreover, the DSL can be used to tune diversity for particular problems but it is not required: unless other approaches where the diversity goal is required, our method can produce diverse instances without a DSL specification.

On the other hand, most approaches using a diversity goal guide the model finder during search for instances, *i.e.*, diversity is enforced during the search. Instead, our approach does not guide the search but distills diverse instances among a set of solutions. Nevertheless, this mode of operation allows it to be very flexible: it is not tied to a particular algorithm or solver, so it can be applied on top of any existing model finder, using it as black box without requiring any modification. This means that it can extend existing model finders to make them diversity-aware.

8.2. Redundancy avoidance

Some methods operate inside the model finder, reducing the number of instances being computed by detecting equivalent instances or discarding instances that are too similar to previous solutions.

We exclude from this discussion methods designed for general-purpose solvers [63,33,32], as (1) they have not been used within model finders and (2) they consider diversity at a lower level of abstraction (*e.g.*, assignments to a boolean formula) where some model-level similarities may be lost (*e.g.*, isomorphic instances may have different bit-vector representations but they are still equivalent). For instance, a related software engineering problem that relies on low-level constraint solvers is finding valid configurations in a software product line. In this context, it has been shown [31] that SAT solvers designed for *uniform sampling* (*i.e.*, computing satisfying assignments that are distributed as close as possible to a uniform distribution) do not achieve a uniform distribution in the set of computed configurations.

Some techniques aim to automatically *detect equivalent solutions* during the analysis in order to avoid exploring them. In the context of boolean satisfiability (SAT), SAT-Modulo Theories (SMT) and Constraint Programming (CP) this notion is called *symmetry breaking* [10,14] and it is achieved by including additional constraints. A different approach to enforce some degree of canonicity is requiring the solution to be *minimal*, either because removing any element makes the solution invalid [19] or because one of the goals of our objective function is minimizing the size of the solution [52,54].

In search-based methods like genetic algorithms [53,54] or simulated annealing [52], similarity among solutions can be detected through a *distance measure*: neighbors that are too close to previously explored solutions can be ignored. Similarly, in graph solvers *graph shape analysis* [55,2] can detect equivalent or similar graphs. However, even though some graph solvers provide support for attributes during model finding [25], they do not include attribute values in the diversity analysis and do not support other features (*e.g.*, relations and witnesses) like the approach presented in this paper.

Finally, model finders can introduce *randomness* [50], such as random selection of the next value to be explored or random restarts that can help explore different areas of the search space. Another take on randomness, *randomized partitioning* [15], shares the goal of classifying terms (partitioning the solution space) but generates the partitions by randomly splitting the domains of model elements. While this approach may be successful in problems with simple and local constraints, it is ineffective when dealing with complex constraints.

While all these methods somehow force the finder to generate more diverse instances by itself, they are specific to particular notations and solvers and thus are less flexible than our approach.

8.3. Clustering of instances

The COMODI tool [56,17] provides several techniques for clustering the object diagrams produced by a UML/OCL model finder. First, it defines a feature vector encoding for object diagrams that captures, for each object, information about attribute values and adjacent objects. And second, it defines a centrality metric (similar to the `pagerank` algorithm of search engines) that measures the importance of each object within the object diagram.

Compared to our method, this approach is specific for object diagrams: it cannot deal with complex features from other modeling notations, such as Alloy's relations or witnesses. Furthermore, the proposed similarity metrics do not consider information about types, structure and attribute values simultaneously: the centrality metric omits attribute values entirely; and the feature vector approach requires defining an ordering for vertices a priori (with results that are very sensitive to this ordering) and does not consider complex topological information about the structure of the object diagram.

8.4. Clustering of meta-models

The last group of works deal with the analysis of meta-model repositories [64,58,65,59]. The methods used by [64,58,65] represent each model as a *feature vector*. Model elements are compared using their identifiers (name) [65], and the relationships between model elements are described using n-grams (paths of n elements inside the model, *e.g.*, pairs of elements for $n = 2$) [64,58,57]. The resulting feature vector can be used to perform group models using hierarchical clustering. Another tool [59] proposes two alternative strategies to compare class diagrams: use the string distance between the textual representations of the serialized models (*e.g.*, in XMI format); or use dedicated tools for model comparison such as EMF Compare.[11]

However, these approaches are intended for analyzing structural diagrams such as class diagrams. Thus, when comparing models they consider features like the names of the class or its number of attributes, rather than the values of those attributes as it happens at the instance level. Therefore, they are not suitable for analyzing instances from a model finder, as they do not support features like attribute values, relations or witnesses.

9. Conclusions

We have presented a method for addressing the lack of diversity among the instances computed by a model finder. Our approach uses clustering to group instances according to their similarity, using information both about topology, types and attribute. The method is solver- and notation-agnostic: it can be applied to model finders using different types of solvers (*e.g.*, SAT, SMT or CP) and even targeting different modeling notations (*e.g.*, UML/OCL or Alloy). An added benefit of this approach is the ability to customize the notion of diversity that should be considered.

This approach is capable of computing meaningful clusters and has an execution time that is negligible with respect to that of the model finder itself. Still, as our diversity computation is an *a posteriori* procedure, it is intended for validation and testing scenarios where model finders are able to find instance solutions with relative ease. In this sense, our approach does not increase the diversity of the model finder output. However, it maximizes diversity by selecting, on behalf of the user, the widest possible variation among the output set.

As future work, we will study the automatic generation of (part of) the diversity configuration directly from the declarative specification. To this end, we can use several heuristics obtained from an analysis of the structure of the model. For instance, we can assume that model elements that are more constrained in the model are also more relevant in terms of diversity than those that are unconstrained. Moreover, we will work on extending the capabilities of the DSL in order to describe relevant diversity properties (*e.g.*, tailored to a particular domain) in a usable and succinct way.

CRediT authorship contribution statement

Robert Clarisó: Conceptualization, Methodology, Software, Visualization, Writing – original draft. **Jordi Cabot:** Conceptualization, Funding acquisition, Methodology, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Funding: This work was partially funded by the H2020 ECSEL Joint Undertaking Projects MegaM@Rt2 (grant number 737494), AIDOaRt (grant agreement number 101007350) and TRANSACT (grant number 101007260); the Spanish Ministry of Economy and Competitiveness through the project ODA (TIN2016-75944-R) and the Agencia Estatal de Investigación through the project LOCOSS (PID2020-114615RB-I00).

We would like to thank the anonymous reviewers for their comments, which have greatly improved the paper.

References

[1] D. Jackson, Software Abstractions: Logic, Language and Analysis, MIT Press, 2006, https://mitpress.mit.edu/books/software-abstractions.
[2] O. Semeráth, A.S. Nagy, D. Varró, A graph solver for the automated generation of consistent domain-specific models, in: International Conference on Software Engineering, ICSE'2018, ACM Press, 2018, pp. 969–980.
[3] M. Petre, UML in practice, in: International Conference on Software Engineering, ICSE'13, IEEE Press, 2013, pp. 722–731.
[4] C.A. González, J. Cabot, Formal verification of static software models in MDE: a systematic review, Inf. Softw. Technol. 56 (8) (2014) 821–838, https://doi.org/10.1016/j.infsof.2014.03.003.
[5] A.D. Brucker, B. Wolff, HOL-OCL: a formal proof environment for UML/OCL, in: Fundamental Approaches to Software Engineering, Springer, 2008, pp. 97–100.
[6] S. Ali, M. Zohaib Iqbal, A. Arcuri, L.C. Briand, Generating test data from OCL constraints with search techniques, IEEE Trans. Softw. Eng. 39 (10) (2013) 1376–1402, https://doi.org/10.1109/TSE.2013.17.
[7] G. Soltana, M. Sabetzadeh, L.C. Briand, Practical model-driven data generation for system testing, ACM Trans. Softw. Eng. Methodol. 29 (2) (Apr. 2020), arXiv:1902.00397.

[11] https://www.eclipse.org/emf/compare/.

[8] G. Rull, C. Farré, A. Queralt, E. Teniente, T. Urpí, AuRUS: explaining the validation of UML/OCL conceptual schemas, Softw. Syst. Model. 14 (2) (2015) 953–980, https://doi.org/10.1007/s10270-013-0350-8.
[9] M. Clavel, M. Egea, ITP/OCL: a rewriting-based validation tool for UML+OCL static class diagrams, in: AMAST'06, Springer, 2006, pp. 368–373.
[10] E. Torlak, D. Jackson, Kodkod: a relational model finder, in: TACAS'07, Springer, Berlin, Heidelberg, 2007, pp. 632–647.
[11] M. Kuhlmann, L. Hamann, M. Gogolla, Extensive validation of OCL models by integrating SAT solving into USE, in: TOOLS'2011, Springer, Berlin, Heidelberg, 2011, pp. 290–306.
[12] Hao Wu, An SMT-based approach for generating coverage oriented metamodel instances, Int. J. Inf. Syst. Model. Des. 7 (3) (2016) 23–50, https://doi.org/10.4018/IJISMD.2016070102, http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/IJISMD.2016070102.
[13] C. Dania, M. Clavel, OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints, in: MODELS'16, ACM Press, USA, 2016, pp. 65–75, http://dl.acm.org/citation.cfm?doid=2976767.2976774.
[14] J. Cabot, R. Clarisó, D. Riera, On the verification of UML/OCL class diagrams using constraint programming, J. Syst. Softw. 93 (2014) 1–23, https://doi.org/10.1016/j.jss.2014.03.023.
[15] E.K. Jackson, G. Simko, J. Sztipanovits, Diversely enumerating system-level architectures, in: International Conference on Embedded Software, EMSOFT'2013, IEEE, 2013, pp. 1–10.
[16] G. Soltana, M. Sabetzadeh, L.C. Briand, Synthetic data generation for statistical testing, in: IEEE/ACM International Conference on Automated Software Engineering, ASE'2017, IEEE, 2017, pp. 872–882.
[17] A. Ferdjoukh, F. Galinier, E. Bourreau, A. Chateau, C. Nebut, Measurement and generation of diversity and meaningfulness in model driven engineering, Int. J. Adv. Softw. 11 (1/2) (2018) 131–146, https://hal-lirmm.ccsd.cnrs.fr/lirmm-02067506.
[18] D. Varró, O. Semeráth, G. Szárnyas, Á. Horváth, Towards the automated generation of consistent, diverse, scalable and realistic graph models, in: Graph Transformation, Specifications, and Nets, Springer, 2018, pp. 285–312.
[19] T. Nelson, S. Saghafi, D.J. Dougherty, K. Fisler, S. Krishnamurthi, Aluminum: principled scenario exploration through minimality, in: International Conference on Software Engineering, ICSE'2013, IEEE, 2013, pp. 232–241.
[20] H. Wu, MaxUSE: a tool for finding achievable constraints and conflicts for inconsistent UML class diagrams, in: International Conference on Integrated Formal Methods, IFM'2017, Springer, 2017, pp. 348–356.
[21] S. Vishwanathan, N.N. Schraudolph, R. Kondor, K.M. Borgwardt, Graph kernels, J. Mach. Learn. Res. 11 (Apr) (2010) 1201–1242, http://www.jmlr.org/papers/v11/vishwanathan10a.html.
[22] S. Ghosh, N. Das, T. Gonçalves, P. Quaresma, M. Kundu, The journey of graph kernels through two decades, Comput. Sci. Rev. 27 (2018) 88–111, https://doi.org/10.1016/J.COSREV.2017.11.002.
[23] R. Clarisó, J. Cabot, Diverse scenario exploration in model finders using graph kernels and clustering, in: International Conference on Rigorous State-Based Methods, ABZ'2020, in: LNCS, vol. 12071, Springer, 2020, pp. 27–43.
[24] O. Semeráth, A.S. Nagy, D. Varró, A graph solver for the automated generation of consistent domain-specific models, in: ICSE '2018, ACM Press, New York, USA, 2018, pp. 969–980, http://dl.acm.org/citation.cfm?doid=3180155.3180186.
[25] O. Semeráth, A.A. Babikian, A. Li, K. Marussy, D. Varró, Automated generation of consistent models with structural and attribute constraints, in: E. Syriani, H.A. Sahraoui, J. de Lara, S. Abrahão (Eds.), MoDELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, ACM, 2020, pp. 187–199.
[26] J. Cabot, M. Gogolla, Object constraint language (OCL): a definitive guide, in: M. Bernardo, V. Cortellessa, A. Pierantonio (Eds.), Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, in: Lecture Notes in Computer Science, vol. 7320, Springer, 2012, pp. 58–90.
[27] A. Azurat, I. Prasetya, A survey on embedding programming logics in a theorem prover, Tech. Rep. UU-CS-2002-007, Institute of Information and Computing Sciences, Utrecht University, 2002.
[28] M. Leuschel, M. Butler, ProB: an automated analysis toolset for the B method, Int. J. Softw. Tools Technol. Transf. 10 (2) (2008) 185–203.
[29] J.-R. Abrial, J.-R. Abrial, The B-Book: Assigning Programs to Meanings, Cambridge University Press, 2005.
[30] M. Leuschel, E. Turner, Visualising larger state spaces in ProB, in: International Conference of B and Z Users, Springer, 2005, pp. 6–23.
[31] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, M. Cordy, Uniform sampling of SAT solutions for configurable systems: are we there yet?, in: IEEE Conference on Software Testing, Validation and Verification, ICST'2019, IEEE, 2019, pp. 240–251.
[32] R. Dutra, K. Laeufer, J. Bachrach, K. Sen, Efficient sampling of SAT solutions for testing, in: International Conference on Software Engineering, ICSE'2018, ACM, 2018, pp. 549–559.
[33] A. Nadel, Generating diverse solutions in SAT, in: International Conference on Theory and Applications of Satisfiability Testing, SAT'2011, Springer, Berlin, Heidelberg, 2011, pp. 287–301.
[34] A. Mougenot, A. Darrasse, X. Blanc, M. Soria, Uniform random generation of huge metamodel instances, in: European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA'2009, in: LNCS, vol. 5562, Springer, 2009, pp. 130–145.
[35] N. Shervashidze, S.V.N. Vishwanathan, T. Petri, K. Mehlhorn, K.M. Borgwardt, Efficient graphlet kernels for large graph comparison, in: D.A.V. Dyk, M. Welling (Eds.), Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, AISTATS 2009, Clearwater Beach, Florida, USA, April 16-18, 2009, in: JMLR Proceedings, vol. 5, JMLR.org, 2009, pp. 488–495, http://proceedings.mlr.press/v5/shervashidze09a.html.
[36] N. Shervashidze, P. Schweitzer, E.J. van Leeuwen, K. Mehlhorn, K.M. Borgwardt, Weisfeiler-Lehman graph kernels, J. Mach. Learn. Res. 12 (2001) 2539–2561, https://dl.acm.org/citation.cfm?id=2078187.
[37] G. Siglidis, G. Nikolentzos, S. Limnios, C. Giatsidis, K. Skianis, M. Vazirgiannis, GraKeL: a graph kernel library in Python, J. Mach. Learn. Res. 21 (2020) 54:1–54:5, https://www.jmlr.org/papers/v21/18-370.html.
[38] G. Li, M. Semerci, B. Yener, M.J. Zaki, Effective graph classification based on topological and label attributes, Stat. Anal. Data Min. 5 (4) (2012) 265–283, https://doi.org/10.1002/sam.11153.
[39] G.K. de Vries, A fast approximation of the Weisfeiler-Lehman graph kernel for rdf data, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2013, pp. 606–621.
[40] A. Narayanan, G. Meng, L. Yang, J. Liu, L. Chen, Contextual Weisfeiler-Lehman graph kernel for malware detection, in: 2016 International Joint Conference on Neural Networks, IJCNN, IEEE, 2016, pp. 4701–4708.
[41] R. Xu, D. Wunsch, Survey of clustering algorithms, IEEE Trans. Neural Netw. 16 (3) (2005) 645–678, https://doi.org/10.1109/TNN.2005.845141.
[42] C.C. Aggarwal, H. Wang, A survey of clustering algorithms for graph data, in: Managing and Mining Graph Data, Springer, 2010, pp. 275–301.
[43] F. Murtagh, P. Contreras, Algorithms for hierarchical clustering: an overview, Wiley Interdiscip. Rev. Data Min. Knowl. Discov. 2 (1) (2012) 86–97.
[44] P.J. Rousseeuw, Silhouettes: a graphical aid to the interpretation and validation of cluster analysis, J. Comput. Appl. Math. 20 (1) (1987) 53–65, https://doi.org/10.1016/0377-0427(87)90125-7.
[45] M. Sugiyama, M.E. Ghisu, F. Llinares-López, K. Borgwardt, graphkernels: R and Python packages for graph comparison, Bioinformatics 34 (3) (2018) 530–532, https://doi.org/10.1093/bioinformatics/btx602, http://www.ncbi.nlm.nih.gov/pubmed/29028902, http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC5860361, https://academic.oup.com/bioinformatics/article/34/3/530/4209994.
[46] A. Cunha, N. Macedo, T. Guimarães, Target oriented relational model finding, in: FASE'14, vol. 8411, Springer-Verlag, New York, Inc., 2014, pp. 17–31.
[47] V. Montaghami, D. Rayside, Bordeaux: a tool for thinking outside the box, in: FASE'17, Springer, 2017, pp. 22–39.

[48] F. Hilken, M. Gogolla, L. Burgueño, A. Vallecillo, Testing models and model transformations using classifying terms, Softw. Syst. Model. 17 (3) (2018) 885–912, https://doi.org/10.1007/s10270-016-0568-3.
[49] O. Semeráth, R. Farkas, G. Bergmann, D. Varró, Diversity of graph models and graph generators in mutation testing, Int. J. Softw. Tools Technol. Transf. 22 (1) (2020) 57–78, https://doi.org/10.1007/s10009-019-00530-6.
[50] A. Ferdjoukh, E. Bourreau, A. Chateau, C. Nebut, A model-driven approach to generate relevant and realistic datasets, in: SEKE, Software Engineering and Knowledge Engineering, 2016, pp. 105–109, http://ksiresearchorg.ipage.com/seke/seke16paper/seke16paper_29.pdf.
[51] G. Szárnyas, Z. Kovári, Á. Salánki, D. Varró, Towards the characterization of realistic models: evaluation of multidisciplinary graph metrics, in: B. Baudry, B. Combemale (Eds.), Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016, ACM, 2016, pp. 87–94, http://dl.acm.org/citation.cfm?id=2976786.
[52] J.J. Cadavid, B. Baudry, H. Sahraoui, Searching the boundaries of a modeling space to test metamodels, in: IEEE International Conference on Software Testing, Verification and Validation, ICST'2012, IEEE, 2012, pp. 131–140.
[53] F. Galinier, E. Bourreau, A. Château, A. Ferdjoukh, C. Nebut, Genetic algorithm to improve diversity in MDE, in: META: Metaheuristics and Nature Inspired Computing, 2016, pp. 171–173, https://hal-lirmm.ccsd.cnrs.fr/lirmm-01397321.
[54] E. Batot, H. Sahraoui, A generic framework for model-set selection for the unification of testing and learning MDE tasks, in: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS'2016, ACM Press, USA, 2016, pp. 374–384.
[55] O. Semeráth, D. Varró, Iterative generation of diverse models for testing specifications of DSL tools, in: 19th International Conference on Fundamental Approaches to Software Engineering, FASE'2018, Springer, Cham, 2018, pp. 227–245.
[56] A. Ferdjoukh, F. Galinier, E. Bourreau, A. Chateau, C. Nebut, Measuring differences to compare sets of models and improve diversity in MDE, in: ICSEA, International Conference on Software Engineering Advances, 2017, http://adel-ferdjoukh.ovh/wp-content/uploads/pdf/icsea17-distances-v9.pdf.
[57] O. Babur, L. Cleophas, T. Verhoeff, M. van den Brand, Towards statistical comparison and analysis of models, in: Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, SCITEPRESS - Science and Technology Publications, 2016, pp. 361–367, http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005799103610367.
[58] O. Babur, L. Cleophas, Using n-grams for the automated clustering of structural models, in: SOFSEM'17, Springer, 2017, pp. 510–524.
[59] F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, A. Pierantonio, Automated clustering of metamodel repositories, in: CAiSE'16, Springer, 2016, pp. 342–358.
[60] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, D. Vrgoč, Foundations of modern query languages for graph databases, ACM Comput. Surv. 50 (5) (2017) 1–40.
[61] D. Varró, A. Balogh, The model transformation language of the VIATRA2 framework, Sci. Comput. Program. 68 (3) (2007) 214–234.
[62] D. Balasubramanian, A. Narayanan, C. van Buskirk, G. Karsai, The graph rewriting and transformation language: GReAT, Electron. Commun. EASST 1 (2007).
[63] S.G. Vadlamudi, S. Kambhampati, A combinatorial search perspective on diverse solution generation, in: AAAI Conference on Artificial Intelligence, AAAI Press, 2016, pp. 776–783, https://dl.acm.org/citation.cfm?id=3015927.
[64] O. Babur, Statistical analysis of large sets of models, in: ASE'16, ACM Press, USA, 2016, pp. 888–891, http://dl.acm.org/citation.cfm?doid=2970276.2975938.
[65] O. Babur, L. Cleophas, M. van den Brand, Hierarchical Clustering of Metamodels for Comparative Analysis and Visualization, Springer, Cham, 2016, pp. 3–18.