Diverse Scenario Exploration in Model Finders Using Graph Kernels and Clustering

Robert Clarisó[1(✉)] and Jordi Cabot[2]

[1] Universitat Oberta de Catalunya (UOC), Barcelona, Spain
rclariso@uoc.edu
[2] ICREA, Barcelona, Spain
jordi.cabot@icrea.cat

Abstract. Complex software systems can be described using modeling notations such as UML/OCL or Alloy. Then, some correctness properties of these systems can be checked using model finders, which compute sample scenarios either fulfilling the desired properties or illustrating potential faults. Such scenarios allow designers to validate, verify and test the system under development.

Nevertheless, when asked to produce several scenarios, model finders tend to produce similar solutions. This lack of diversity impairs their effectiveness as testing or validation assets. To solve this problem, we propose the use of *graph kernels*, a family of methods for computing the (dis)similarity among pairs of graphs. With this metric, it is possible to *cluster* scenarios effectively, improving the usability of model finders and making testing and validation more efficient.

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

1 Introduction

The structure and behavior of a software system can be described by means of *software models*, using notations such as Alloy [10], graph-based formalisms [20] or UML/OCL [17]. These notations describe software systems at a high level of abstraction, hiding implementation details while preserving its salient features. Analysing 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* [8], tools capable of computing *instances* of a model that satisfy a set of constraints and properties of interest. Each model finder targets

This work is partially funded by the H2020 ECSEL Joint Undertaking Project "MegaM@Rt2: MegaModelling at Runtime" (737494) and the Spanish Ministry of Economy and Competitivity through the project "Open Data for All: an API-based infrastructure for exploiting online data sources" (TIN2016-75944-R).

A. Raschke et al. (Eds.): ABZ 2020, LNCS 12071, pp. 27–43, 2020.
https://doi.org/10.1007/978-3-030-48077-6_3

a particular modeling notation and uses a different reasoning engine, like search-based methods [1,24], SAT [10], SMT [24,28] or constraint programming [3].

For verification purposes, it is usually enough to search for one instance, which either proves or disproves the property of interest. 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 [11]. 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.

Nevertheless, most model finders focus on efficiency and expressiveness of the input modeling notation, so few of them ensure diversity of the generated instances [6,11,20,23,26]. 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 [11,20,26] or dynamic properties [6,11,23,24]) and cannot benefit from additional features provided by others model finders (*e.g.*, computation of minimal instances [16] or support for max-satisfiability [28]).

This paper proposes a method for distilling diverse instances in the model finder output based on the use of *clustering*. Instances are classified into categories according to their *similarity*, which is calculated using information about their *structure* (the existing objects and the links between them), *typing* (the specific type of each object) and *attribute values*. This calculation is based on the use of *graph kernels*, a family of methods for computing distances among graphs. Selecting a representative instance from each category ensures diversity while reducing testing and validation time, as redundant instances can be safely discarded. As a drawback, this method does not *force* the model finder to look for diverse instances, it only distills the most diverse ones.

Compared with related works, our approach offers the following advantages:

- It is independent of the solver used by the model finder (SAT, SMT, ...) and the modeling notation being analyzed (Alloy, UML/OCL, ...).
- It does not require manual intervention from the designer to define what kind of instances are "relevant" or when two instances are "similar".
- The similarity computation can be customized, *e.g.*, by selecting a trade-off between precision and accuracy.

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 three steps of our method: the *abstraction* process for transforming instances into graphs (Sect. 3); *graph kernels* (Sect. 4), the framework for computing similarities among graphs; and *clustering* algorithms that can use this similarity to build groups of related instances (Sect. 5). Section 6 presents some experimental results of the application of this method. After that,

Sect. 7 describes previous work on diversity and model finding. Finally, Sect. 8 outlines the conclusions and lines 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, and 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.

The method can be divided into three steps:

1. **Graph abstraction**: 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. **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 $n \times n$ matrix S where each cell S_{ij} provides information about the similarity between graphs i and j.
3. **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.

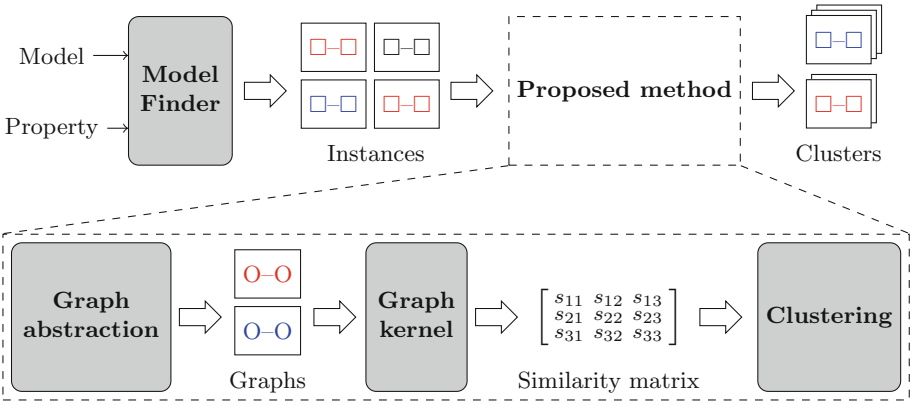


Fig. 1. Overview of the method presented in this paper.

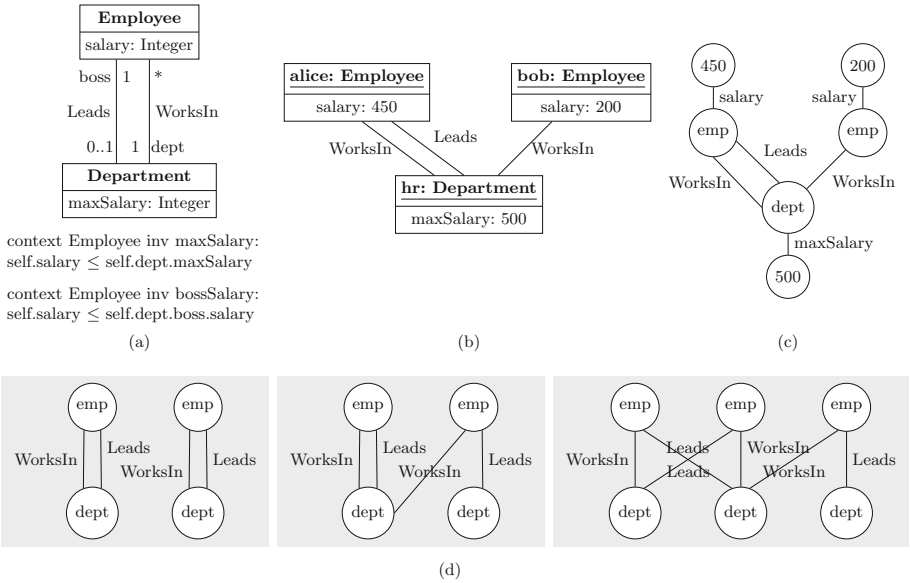


Fig. 2. Motivating example: (a) UML/OCL class diagram; (b) Sample instance; (c) Encoding of the instance as a labeled graph; (d) Graph shapes of the three clusters.

To be usable in practice, this model should be *strongly satisfiable* [3]: 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 [12] to generate 25 valid instances for this model. By manually inspecting these instances, we can easily realize that most of them are very similar. 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). 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).

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. This is a corner case worth studying, to decide whether it should actually be allowed or it is a mistake in the model that needs to be fixed.

The following sections describe the different phases of our approach in detail.

3 Graph Abstraction

Depending on the model finder, instances have a different structure, *e.g.*, an object diagram, an enriched graph or a set of tuples. In order to take advantage of off-the-shelf graph comparison algorithms, we translate these instances into labeled undirected graphs. To this end, we define the vertices, edges and labels in the graph in terms of the original 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 that can, later on, help to establish whether a pair of vertices from two different graphs can be considered "equivalent".

The complexity of this step depends on the kind of output 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 straightforward according to this pattern: objects and attributes becomes vertices, links become edges, and types and attribute values become labels. 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.

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` or `lone` (zero or 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 of constraint: *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 considered for each signature.

Command `check` searches for a counterexample of an assertion, while command `run` searches for an example of a predicate.

Alloy Snapshots. Executing a command with the Alloy Analyzer may yield two outcomes: either no instance within the scope satisfies the constraints or an instance has been found. 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 computes for which specific value of the quantified variable (the witness) the property holds.

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 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 where 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.

Figure 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`). Figure 3(b) and (c) show one sample Alloy instance in textual and graphical format. The corresponding graph

[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.

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.

Abstraction and Diversity. Some approaches aimed at achieving diversity use *uniform sampling* [5,14,15,18] 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 [6,24]. In the following, we discuss how this information about the desired type of diversity can be integrated in the graph abstraction process with very few changes.

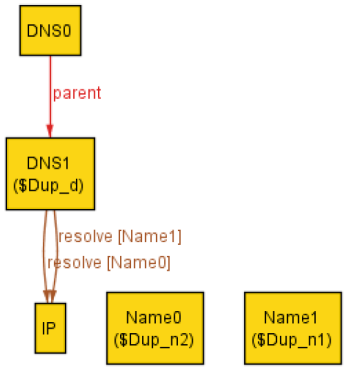
For example, let us consider the specification of a banking system. From our domain knowledge, it seems reasonable to think that the name of the owner an account is not very relevant: if there are 10 clients in our system, the fact

```
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
```

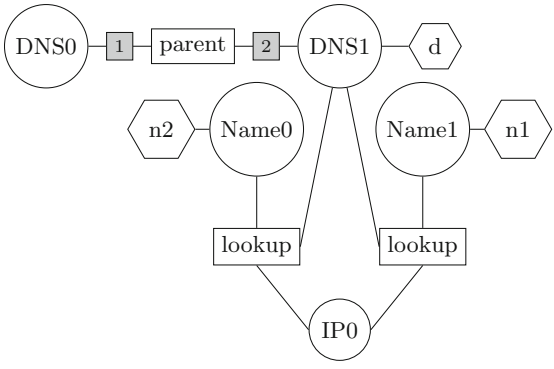
(a)

```
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.

that all of them are called "John Smith" might not be problematic. Thus, the name of the owner could be abstracted away in our graph representation, *i.e.* remove from the graph the vertices related to this particular attribute. On the other hand, focusing on the balance of an account, we might be interested in considering accounts with a positive, negative and zero balance. In this case, we are not interested in specific values for the balance, only if they fit in these three categories. In our graph abstraction, this situation can be modeled by using these categories (instead of the integer value) as the label for the vertex.

4 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 checking for *isomorphism*[2] 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 [7,27], 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. In this work, we have used the Weisfeiler-Lehman kernel [22], as it has been shown to provide good precision with an efficient computation in a variety of domains [13,22].

Algorithm 1 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 [22]. 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.

[2] 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 $(f(x), f(y)) \in E_2$.

```
1  Function WLKernel(G1, G2, h)                          // Weisfeiler-Lehman graph kernel
       input  : G1, G2: a pair of labeled graphs; h : an integer (the tree height)
       output: A distance measure between G1 and G2
2      freq1 ← WLTest(G1, h);                            // frequency of each label in G1
3      freq2 ← WLTest(G2, h);                            // frequency of each label in G2
4      distance ← 0;                                     // distance = difference among frequencies
5      foreach label lab do
6          distance ← distance + |freq1[lab] − freq2[lab]|;
7      return distance;

8  Function WLTest(G, h)                                 // Weisfeiler-Lehman isomorphism test
       input  : G: a labeled graph G = (V, E); h : an integer (the tree height)
       output: A map counting the frequency of labels in G
9      // Initially all labels x have frequency[x] = 0
10     foreach vertex v ∈ V(G) do
11         label(0,v) ← label of v in G;
12         frequency[label(0,v)] ← frequency[label(0,v)] + 1
13     for i ← 1 to h do
14         foreach vertex v ∈ V(G) do
15             adjacentLabels ← labels(i-1, neighbours(v, G));
16             // signature = my label + sorted labels of adjacent vertices
17             signature ← append(label(i-1,v), sort(adjacentLabels));
18             // Assign an integer label that summarizes signature
19             // Two equal signature should always receive the same label
20             // Compressed labels not reused in the next iterations
21             label(i, v) ← compressLabels(signature) ;
22             frequency[label(i,v)] ← frequency[label(i,v)] + 1
23     return frequency;
```

Algorithm 1: Pseudocode for the Weisfeiler-Lehman graph kernel [22].

5 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".

Algorithm Selection. Several algorithms have been proposed for this task [29]. 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* 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 algorithms 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 abstracting the outputs of a model finder. The number of target clusters is unknown a priori and, as discussed in the previous section, we will be using a similarity metric. Given this setting, we have chosen hierarchical clustering.

Choice of Number of Clusters. Hierarchical clustering computes a hierarchical structure called *dendogram*, 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.

In this work, we have used the *silhouette coefficient* [19], 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.

6 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[3]. Among them, we have chosen examples dealing with the generation of examples or counterexamples, rather than proving their absence. These type 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 1 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 abstraction. 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

[3] https://github.com/AlloyTools/models.

Table 1. Summary of the models analyzed with the Alloy Analyzer.

Model	Domain	Sig	Field	Fact	Pred
chord-bug-model	Chord distributed hastable lookup protocol	4	8	3	15
file-system	Generic file system	7	4	0	3
firewire	Leader election in the 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

performs the clustering. This script takes advantage of existing libraries for representing graphs (the `igraph` package[4]), similarity analysis among graphs (the `graphkernels` package[5]) and clustering (the `cluster` package[6]).

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.

Table 2. Experimental results.

			Model	Execution time				Output	
Model	Scope	Inst	finding	Abst	Kern	Clust	Total	# Cl	Sil
chord-bug-model	2	52	498 ms	169 ms	90 ms	30 ms	289 ms	5	0.31
file-system	5	100	825 ms	165 ms	180 ms	30 ms	375 ms	3	0.99
firewire	2–7	100	1474 ms	209 ms	180 ms	40 ms	429 ms	3	0.76
flip-flop	10	100	652 ms	203 ms	180 ms	50 ms	433 ms	2	0.04
genealogy	6	100	830 ms	129 ms	140 ms	50 ms	319 ms	33	0.45
grandpa	4	48	554 ms	88 ms	70 ms	40 ms	198 ms	2	0.96
life	3–6	100	1681 ms	283 ms	180 ms	40 ms	503 ms	14	0.30
philosophers	4	100	1539 ms	157 ms	160 ms	40 ms	357 ms	2	0.30
railway	1–4	100	735 ms	179 ms	170 ms	30 ms	379 ms	50	0.46
reset-flip-flop	10	100	672 ms	250 ms	160 ms	40 ms	450 ms	14	0.48

[4] https://igraph.org.
[5] https://cran.r-project.org/package=graphkernels.
[6] https://cran.r-project.org/package=cluster.

Results and Discussion. Table 2 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. Finally, we list the optimal number of clusters (**# Cl**) identified by our method and the silhouette coefficient (**Sil**). As mentioned in Sect. 5, 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 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) 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 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.

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.

7 Related Work

Several works have considered how to improve the diversity in the output of model finders, *e.g.*, [6,9,11,20,23,26]. We will classify them according to two criteria: (i) how diversity is specified by the designer and (ii) how it is achieved.

We exclude from this discussion all methods designed for general-purpose solvers [5,15,25], as they have not been used within model finders and 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 with different bit-vector representations 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 [18] 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.

Definition of Diversity. The designer has different ways to specify the desired notion of diversity. Some methods [6,23] need to be given a *probability distribution* that the output instances should follow. Otherwise, the designer can partition the universe of instances by defining predicates called *classifying terms* [9]. For instance, for an attribute the designer may only be interested in its sign (positive, negative or zero), defining 3 partitions. Diversity is then achieved by finding instances that cover each partition.

Meanwhile, other methods such as [11,26] or the one proposed in this paper 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. Nevertheless, in our case, the designer has some degree of control over the desired type of diversity by adapting the graph abstraction process, as explained in Sect. 3.

Implementation of Diversity. Most methods operate inside the model finder, reducing the number of instances being computed in different ways.

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* [3,10] and it is achieved by including additional constraints a priori. These constraints can also be added dynamically each time a new instance is found [9,23], to forbid exploring equivalent instances in the future. Another way to avoid exploring equivalent instances is requiring the solution to be *minimal* [2,4,16].

In search-based methods like genetic algorithms [2] or simulated annealing [4], 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* [20,21] can detect equivalent or similar graphs. Nevertheless, this approach does not support features like attributes, relations or witnesses like the approach presented in this paper.

Moreover, model finders can introduce *randomness* [6], 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* [11], 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.

Finally, the COMODI tool [6] 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 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 does not consider topological information about the structure of the object diagram.

8 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).

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 plan to define custom kernels for comparing instances that take into account specific characteristics of the input model. For instance, the invariants and multiplicities in the model can be used to identify which model elements are more constrained: this is where diversity is most relevant, rather than elements where we are free to choose almost any value. Also, we plan to look into combining graph kernels with topological and label features [13] that can improve the quality of the similarity analysis. Finally, we will consider strategies for tailoring the graph abstraction to particular problems and domains.

References

1. Ali, S., Zohaib Iqbal, M., Arcuri, A., Briand, L.C.: Generating test data from OCL constraints with search techniques. IEEE Trans. Softw. Eng. **39**(10), 1376–1402 (2013). https://doi.org/10.1109/TSE.2013.17
2. Batot, E., Sahraoui, H.: 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), pp. 374–384. ACM Press, New York (2016). https://doi.org/10.1145/2976767.2976785
3. Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. J. Syst. Softw. **93**, 1–23 (2014). https://doi.org/10.1016/j.jss.2014.03.023
4. Cadavid, J.J., Baudry, B., Sahraoui, H.: Searching the boundaries of a modeling space to test metamodels. In: IEEE International Conference on Software Testing, Verification and Validation (ICST 2012), pp. 131–140. IEEE (2012). https://doi.org/10.1109/ICST.2012.93
5. Dutra, R., Laeufer, K., Bachrach, J., Sen, K.: Efficient sampling of SAT solutions for testing. In: International Conference on Software Engineering (ICSE 2018), pp. 549–559. ACM (2018). https://doi.org/10.1145/3180155.3180248
6. Ferdjoukh, A., Galinier, F., Bourreau, E., Chateau, A., Nebut, C.: Measurement and generation of diversity and meaningfulness in model driven engineering. Int. J. Adv. Softw. **11**(1/2), 131–146 (2018). https://hal-lirmm.ccsd.cnrs.fr/lirmm-02067506
7. Ghosh, S., Das, N., Gonçalves, T., Quaresma, P., Kundu, M.: The journey of graph kernels through two decades. Comput. Sci. Rev. **27**, 88–111 (2018). https://doi.org/10.1016/J.COSREV.2017.11.002
8. González, C.A., Cabot, J.: Formal verification of static software models in MDE: a systematic review. Inf. Softw. Technol. **56**(8), 821–838 (2014). https://doi.org/10.1016/j.infsof.2014.03.003
9. Hilken, F., Gogolla, M., Burgueño, L., Vallecillo, A.: Testing models and model transformations using classifying terms. Softw. Syst. Modeling **17**(3), 885–912 (2016). https://doi.org/10.1007/s10270-016-0568-3
10. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press, Cambridge (2006). https://mitpress.mit.edu/books/software-abstractions
11. Jackson, E.K., Simko, G., Sztipanovits, J.: Diversely enumerating system-level architectures. In: International Conference on Embedded Software (EMSOFT 2013), pp. 1–10. IEEE, September 2013. https://doi.org/10.1109/EMSOFT.2013.6658589

12. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating SAT solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 290–306. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21952-8_21
13. Li, G., Semerci, M., Yener, B., Zaki, M.J.: Effective graph classification based on topological and label attributes. Stat. Anal. Data Mining **5**(4), 265–283 (2012). https://doi.org/10.1002/sam.11153
14. Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform random generation of huge metamodel instances. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 130–145. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02674-4_10
15. Nadel, A.: Generating diverse solutions in SAT. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 287–301. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21581-0_23
16. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: principled scenario exploration through minimality. In: International Conference on Software Engineering (ICSE 2013), pp. 232–241. IEEE, May 2013. https://doi.org/10.1109/ICSE.2013.6606569
17. Petre, M.: UML in practice. In: International Conference on Software Engineering (ICSE 2013), pp. 722–731. IEEE Press (2013). https://doi.org/10.1109/ICSE.2013.6606618
18. Plazar, Q., Acher, M., Perrouin, G., Devroey, X., Cordy, M.: Uniform sampling of SAT solutions for configurable systems: are we there yet? In: IEEE Conference on Software Testing, Validation and Verification (ICST 2019), pp. 240–251. IEEE (2019). https://doi.org/10.1109/ICST.2019.00032
19. Rousseeuw, P.J.: Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. J. Comput. Appl. Math. **20**(1), 53–65 (1987). https://doi.org/10.1016/0377-0427(87)90125-7
20. Semeráth, O., Nagy, A.S., Varró, D.: A graph solver for the automated generation of consistent domain-specific models. In: International Conference on Software Engineering (ICSE 2018), pp. 969–980. ACM Press (2018). https://doi.org/10.1145/3180155.3180186
21. Semeráth, O., Varró, D.: Iterative generation of diverse models for testing specifications of DSL tools. In: Russo, A., Schürr, A. (eds.) FASE 2018. LNCS, vol. 10802, pp. 227–245. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89363-1_13
22. Shervashidze, N., Schweitzer, P., van Leeuwen, E.J., Mehlhorn, K., Borgwardt, K.M.: Weisfeiler-Lehman graph kernels. J. Mach. Learn. Res. **12**, 2539–2561 (2001). https://dl.acm.org/citation.cfm?id=2078187
23. Soltana, G., Sabetzadeh, M., Briand, L.C.: Synthetic data generation for statistical testing. In: IEEE/ACM International Conference on Automated Software Engineering (ASE 2017), pp. 872–882. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115698
24. Soltana, G., Sabetzadeh, M., Briand, L.C.: Practical model-driven data generation for system testing. ACM Transactions on Software Engineering and Methodology (2020, to appear). http://arxiv.org/abs/1902.00397
25. Vadlamudi, S.G., Kambhampati, S.: A combinatorial search perspective on diverse solution generation. In: AAAI Conference on Artificial Intelligence, pp. 776–783. AAAI Press (2016). https://dl.acm.org/citation.cfm?id=3015927

26. Varró, D., Semeráth, O., Szárnyas, G., Horváth, Á.: Towards the automated generation of consistent, diverse, scalable and realistic graph models. In: Heckel, R., Taentzer, G. (eds.) Graph Transformation, Specifications, and Nets. LNCS, vol. 10800, pp. 285–312. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75396-6_16
27. Vishwanathan, S., Schraudolph, N.N., Kondor, R., Borgwardt, K.M.: Graph kernels. J. Mach. Learn. Res. **11**(Apr), 1201–1242 (2010). http://www.jmlr.org/papers/v11/vishwanathan10a.html
28. Wu, H.: MaxUSE: a tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 348–356. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_23
29. Xu, R., Wunsch, D.: Survey of clustering algorithms. IEEE Trans. Neural Netw. **16**(3), 645–678 (2005). https://doi.org/10.1109/TNN.2005.845141