Towards the Characterization of Realistic Model Generators using Graph Neural Networks

José Antonio Hernández López
Department of Computer Science and Systems
Universidad de Murcia
Murcia, Spain
joseantonio.hernandez6@um.es

Jesús Sánchez Cuadrado
Department of Computer Science and Systems
Universidad de Murcia
Murcia, Spain
jesusc@um.es

Abstract—The automatic generation of software models is an important element in many software and systems engineering scenarios such as software tool certification, validation of cyber-physical systems, or benchmarking graph databases. Several model generators are nowadays available, but the topic of whether they generate realistic models has been little studied. The state-of-the-art approach to check the realistic property in software models is to rely on simple comparisons using graph metrics and statistics. This generates a bottleneck due to the compression of all the information contained in the model into a small set of metrics. Furthermore, there is a lack of interpretation in these approaches since there are no hints of why the generated models are not realistic. Therefore, in this paper, we tackle the problem of assessing how realistic a generator is by mapping it to a classification problem in which a Graph Neural Network (GNN) will be trained to distinguish between the two sets of models (real and synthetic ones). Then, to assess how realistic a generator is we perform the Classifier Two-Sample Test (C2ST). Our approach allows for interpretation of the results by inspecting the attention layer of the GNN. We use our approach to assess four state-of-the-art model generators applied to three different domains. The results show that none of the generators can be considered realistic.

Index Terms—Model generators, Realistic models, Graph neural networks, Two-Sample Test

I. INTRODUCTION

Model generators aim to create synthetic models automatically by taking one or more input parameters to configure the expected bounds or shape of the generated models. These types of tools can be applied to many areas of software and system engineering such as the testing and benchmarking of graph databases [1], [2], to create complex test stubs in the object-oriented field [1], [3] or automated synthesis of prototypical test contexts in the assurance of smart cyber-physical systems [1], [4], [5].

Recent works [1], [5] have established four properties that a model generator should satisfy: 1) consistency (the generator creates consistent models which satisfy all well-formedness constraints), 2) diversity (the generated models include a sufficiently wide variety of shapes [6]), 3) scalable (with respect to the size of a generated model) and 4) realistic (models generated cannot be distinguished from the real ones). In particular, a generator is structurally realistic if the set of generated models cannot be distinguished from the real ones just by looking at the typed graph structure (ignoring the attribute values) [5], [6]. Currently, a set of graph metrics and statistics (such as out-degree, dimensional degree, multiplex participation coefficient, etc) are used to assess whether a set of models can be considered similar to a dataset of realistic models [5], [7], [8]. This technique has three important shortcomings. Firstly, summarizing an entire graph model into a set of graph metrics causes an information loss. Secondly, a subset of graph statistics has to be chosen to perform the assessment, but not all metrics are equally effective to perform this task [7]. Thirdly, this approach is not interpretable, in the sense that it does not give us hints to determine why the generator is not realistic.

In this paper, we address the task of determining whether a model generator is realistic using a different technique, which overcomes these aforementioned issues. Our approach follows this idea: given two set of models, one generated by a given generator and the other composed by real models. Under the supposition that the generator is realistic (i.e., the synthetic models are realistic), if we build a Graph Neural Network (GNN) [9] trained to distinguish between these two sets, it will not be able to achieve a good performance since the classification problem is impossible. In order to evaluate how realistic a generator is, we use the non-parametric test called Classifier Two-sample Test (C2ST) [10]. The test tells us whether the GNN actually distinguished between real and generated models. Following this approach, we have assessed how realistic four state-of-the-art generators are in three domains.

Altogether, this paper presents a novel approach for assessing model generators, which is more robust than previous approaches and has the additional advantage of being interpretable (i.e., by inspecting the weights of the attention layer of the GNN). Moreover, to the best of our knowledge, this is the first work that applies GNN to software models. Thus, our proposed GNN architecture can be adapted to face other model classification problems (e.g., meta-model classification [11], [12], UML classification [13], etc).

Organization. Section II gives a brief explanation of the technical background that underlies our approach, which is described in detail in Sect. III. In Sect. IV, we report our experiments assessing four state-of-the-art generators for three
domains. Section V presents the main contributions of this paper and the limitations of our method. Finally, Sect. VI discusses the related work and Sect. VII concludes.

II. BACKGROUND

This section presents background information about the elements of our approach. As a running example to illustrate the paper, let us consider a scenario in which we want to generate Ecore models [14]. An excerpt of the Ecore meta-model is presented in Fig. 1, and an instance of this meta-model representing a Finite State Machine (FSM) is depicted in Fig. 2.

A. Models as graphs

A model can be seen as a graph [7], [15]–[17]. In our approach, models will be mapped to a labeled and directed multigraphs in which nodes and edges are labeled. Therefore, we consider $G = (V, E, f, \mu_V, \mu_E)$, where $V$ is the set of nodes, $E$ is the set of edges, $f : E \rightarrow V \times V$ establishes the source and the target for each edge, and $\mu_V$, $\mu_E$ are the labelling functions (i.e., given a node or an edge, they provide its corresponding label). Given a model, the mapping is performed as follows:

- Each object of the model is mapped into a node in $V$ and it is labeled with the name of the class that the object belongs to.
- Each edge in $E$ is associated with a reference in the model and it is labeled with the name of the reference. If a reference does not have an opposite reference, we add another edge whose label is the concatenation of the reference name and $\text{inv}$.
- In this work we do not consider attributes, since we are only interested on the graph structure.

For instance, the model in Fig. 2 is transformed into the graph in Fig. 3. This graph has 8 nodes (one per different object): 3 nodes/objects of type $E\text{Class}$, 4 nodes/objects of type $E\text{Reference}$ and 1 node/object of type $E\text{Package}$. Edges labeled with $e\text{Type inv}$ are added since the reference $e\text{Type}$ that connects the meta-classes $E\text{TypedElement}$ and $E\text{Classifier}$ does not have an opposite (Fig. 1).

B. Realistic model generators

Models generators aim to generate software models automatically. In this work we have considered four different model generators: EMF Random Instantiator [18], VIATRA [1], Alloy [19], [20] and Random EMF [21]. Depending on the generator, it will receive as input one or more of the following parameters:

- A meta-model $M$ that the generated models will conform to.
- A set of well-formedness constraints $\Psi$. This set refers to a set of conditions that the generated models must verify. For instance, in Ecore, no cycles in hierarchy is a constraint.
- A scope that controls the size of the output e.g. the number of objects that the output models will have.
- A set of rules $\mathcal{G}$ that guides the model generation.

For example, VIATRA generator [1] receives a meta-model, a set of constraints and a scope. However, RandomEMF [21] receives a meta-model and a set of rules that guide the process of the model generation.

On the other hand, Varró et al. [5] presented the Graph Model Generation Challenge in which four properties that a generator should satisfy are established: a model generator has to be consistent, scalable, diverse, and realistic. In this paper we will focus on this last property. In particular, we will deal with the structurally realistic property (studied in [5], [7], [8]). A generator is structurally realistic if the generated
models cannot be distinguished from real models just by looking at the typed graph structure i.e., ignoring attributes. In this work, we refine the notion of realistic generator as follows: Given a classifier trained with a sample of realistic and synthetic models, a generator is realistic if the classifier does not achieve a good performance distinguishing between synthetic and real models.

C. Graph neural networks

A Graph Neural Network (GNN) or Graph Convolutional Neural Network (GCNN) [9] is a type of neural network that receives as input the nodes of a graph and generates node embeddings based on local network neighborhoods. More concretely, given a node \( v \in V \) and its initial vector representation \( x_v \), a GNN of \( L \) layers consist on the calculus of the final node embedding in this way:

\[
h_v^0 = x_v
\]

\[
h_v^l = g_l(h_v^{l-1}, \{h_w^{l-1} : w \in \mathcal{N}(v)\})
\]

for \( l = 1, \ldots, L \), \( \mathcal{N}(v) = \{w \in V|\exists e \text{ such that } f(e) = (w, v)\} \) (neighborhood of \( v \)) and \( g_l \) is a non-linear function that computes \( h_v^l \) using the vectors \( h_w^{l-1} \) and \( \{h_w^{l-1} : w \in \mathcal{N}(v)\} \). Therefore, the final embedding \( h_v^L \) will have information of all neighbors in the \( L \)-hop neighborhood of \( v \). To summarize, a GNN receives a set of nodes \( V \) in a graph and outputs node embedding vectors \( \{h_v^L\}_{v \in V} \) that contain information of the \( L \)-hop neighborhood of each node.

The application of this type of layer can be seen as a message passing layer in which the embedding of a node is calculated by using the messages that it receives from its neighborhood. For example, Fig. 4 shows the application of a graph convolutional layer on our running example. Thick arrows are the messages that the vector \( h_v^l \) receives from its \( 1 \) hop neighborhood. For example, Fig. 4 shows the application of a graph convolutional layer on our running example. Thick arrows are the messages that the vector \( h_v^l \) receives from its \( 1 \) hop neighborhood. Therefore, the new embedding associated to the node \( v \) (i.e., \( h_v^l \)) is calculated by applying \( g_l \) to its last embedding \( h_v^{l-1} \) and the embeddings of its neighborhood \( h_{w_1}^{l-1}, h_{w_2}^{l-1}, h_{w_3}^{l-1}, h_{w_4}^{l-1} \) (dotted arrows).

GNNs are used in problems such as node classification, link prediction or graph classification [22]. We are interested in the last one. In this task, the embedding vectors \( \{h_v^L\}_{v \in V} \) are normally summarized in one vector:

\[
h_G = \text{AGG} \{h_v^L : v \in V\}
\]

where \( \text{AGG} \) is an order-invariant operation such as vector average, max operation, attention mechanisms, etc. This last vector represents the entire graph. Finally, \( h_G \) will be the input to a fully connected neural network which performs the classification using the sigmoid activation in its last layer (or softmax if the problem has more than two classes).

D. Classifier Two-Sample Test

The main aim of a two-sample test is to assess whether two samples \( x_1, \ldots, x_n \sim P \) and \( y_1, \ldots, y_n \sim Q \) come from the same distribution, i.e., whether \( P = Q \). In particular, the following test is considered:

\[
H_0 : P = Q, \\
H_1 : P \neq Q
\]

The Classifier Two-Sample Test (C2ST) [10] tries to solve this problem using this idea: under \( H_0 \) (i.e., both distributions \( P \) and \( Q \) are the same), if we train a classifier to distinguish between \( P \) and \( Q \) (using the original samples), the classifier will not be able to achieve a good performance because both samples contain similar elements as they come from the same distribution. Thus, the expected accuracy (proportion of correctly predicted data samples) will be \( \sim 0.5 \) (near chance-level).

In details using the notation of [10], let us consider the dataset constructed using the samples \( \{x_i\}_{i=1}^n \) and \( \{y_i\}_{i=1}^n \) and associating a label to each sample (0 if it comes from \( P \) and 1 if it comes from \( Q \)):

\[
\mathcal{D} = \{(x_i, 0)\}_{i=1}^n \cup \{(y_i, 1)\}_{i=1}^n = \{(z_i, l_i)\}_{i=1}^{2n}.
\]

\( \mathcal{D} \) is randomly split into \( \mathcal{D}_{\text{train}} \) and \( \mathcal{D}_{\text{test}} \). After that, a classifier is trained using \( \mathcal{D}_{\text{train}} \) to distinguish between labels 0 and 1. This model has the form \( h(z) = P(l = 1|z) \) and it estimates the probability of a sample belonging to label 1. Then, it is evaluated on \( \mathcal{D}_{\text{test}} \). The accuracy, \( \hat{t} \), will be the statistic used to perform the hypothesis test:

\[
\hat{t} = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{(z_i, l_i) \in \mathcal{D}_{\text{test}}} \mathcal{I}(h(z_i) > 0.5) = l_i,
\]

where \( \mathcal{I} \) denotes the indicator function. Basically, \( \hat{t} \) is the proportion of correctly predicted data samples in the test set. The idea of C2ST is that, under \( H_0 \), \( \hat{t} \) should be close to 0.5, but if we assume \( H_1 \), \( \hat{t} \) should be greater than 0.5.

Following the procedure of the framework of statistical hypothesis testing [23], once the statistic \( \hat{t} \) has been computed, we are interested in the \( p \)-value, i.e., \( P(T \geq \hat{t}|H_0) \). Under \( H_0 \), the null distribution can be approximated by
Therefore, some of the generated output models could not be valid. We can see this generator as the following distribution over models (applying the law of total probability):

\[
P_{\text{gen}}(M) = \sum_{(o,d) \in O} P(o)P_{\text{RANDOM}}(M|M,o,d).
\]

Therefore, to sample a model from this distribution we have to sample \((o,d) \sim P\) and then use the generator to sample \(M\). In practice, the distribution \(P\) is approximated using pairs of \((o,d)\) from reals models (see next section).

2) VIATRA: This generator \([1]\) receives a meta-model \(\mathcal{M}\), the scope (normally the number of objects, \(o\)) of the output models and a set of well-formedness constraints \(\Psi\). VIATRA maps the generation problem into a search problem of consistent models. It uses a back-end graph solver which makes this generator scalable. Similar to the EMF random instantiator, VIATRA can be seen as the following distribution:

\[
P_{\text{gen}}(M) = \sum_{o \in \mathcal{O}} P(o)P_{\text{VIATRA}}(M|\mathcal{M},\Psi,o).
\]

Therefore, to sample a model from this distribution we have to sample \(o \sim P\) and then use VIATRA to sample \(M\).

3) Alloy: Alloy Analyzer \([19], [20]\) is a SAT-based model finder that can be used as generator of consistent models. It receives the same input as VIATRA and maps the generation problem into logic problem. Therefore, this generator can be seen as the following distribution:

\[
P_{\text{gen}}(M) = \sum_{o \in \mathcal{O}} P(o)P_{\text{Alloy}}(M|\mathcal{M},\Psi,o).
\]

When sampling from \(P_{\text{Alloy}}(M|\mathcal{M},\Psi,o)\), we add a random amount of extra true statements (as it is done in \([8], [24]\)) to prevent the solver from running deterministically.

4) RandomEMF (rEMF): This generator is a rule-based generator in which the generation process is driven by user-defined rules. It belongs to the category of generators that are based on formal and graph grammars \([21]\). There are two types of random rules, which are illustrated in Fig. 6. In this example, the first rule (called Package) corresponds to the root rule, and it indicates that an object of type EPackage must be generated and the number of its children classifiers must follow a negative binomial distribution with parameters 1 and 0.9784. The second rule is an alternative rule, and it can randomly derive in one of these three rules: Enum, DataType or Class with priorities 1, 4 and 22 respectively.

\[
\begin{align*}
\text{Package: } & \text{EPackage } \rightarrow \text{ eClassifiers } \rightarrow \text{Classifier}\#\text{Binomial}(1,0.9784) \\
\text{rule 1} & \\
\text{alter } & \text{Classifier } \rightarrow \text{Enum}\#1 \mid \text{DataType}\#4 \mid \text{Class}\#22 \\
\text{rule 2} & \\
\end{align*}
\]

Fig. 6. Rules in RandomEMF.
Since this generator receives a meta-model and a set of grammar rules \( \mathcal{G} \), it can be seen as the following distribution over models:

\[
P_{\text{gen}}(M) = P_{\text{rEMF}}(M|\mathcal{M}, \mathcal{G})
\]

Similar to EMF random instatiator, this generator does not support well-formedness constraints. The set of grammar rules must be designed to enforce the generation of consistent models as much as possible. In our experiments, when designing the rules we have attempted to respect the majority of constraints. However, due to the limitations of the RandomEMF grammar definition language, we could not assure all constraints to be respected in the generated output models. For instance, in Ecore, there is no guarantee that the generated models will respect the restriction of no cycles in hierarchy.

### B. Parameter estimation

To sample models from \( P_{\text{gen}} \) we need to estimate or approximate its parameters from a set of realistic models. This process is different depending on the parameters of the generators.

1) Parameter estimation for VIATRA and Alloy: To approximate \( P(o) \) we use the Kernel Density Estimation (KDE), which is a well-known method to estimate the density function of a distribution. Let us suppose that we have a set of samples \( \{o_1, \ldots, o_n\} \) corresponding to the number of objects in each realistic model. As estimation of the density function KDE takes:

\[
\hat{f}_{h,K}(o) = \frac{1}{nh} \sum_{i=1}^{n} K \left( \frac{o - o_i}{h} \right).
\]

Where \( K \) is a kernel function and \( h > 0 \) is called bandwidth. \( K \) and \( h \) are hyperparameters and they are chosen using cross-validation over the set of samples. Once these hyperparameters are fixed, we can sample from \( \hat{f}_{h,K} \) and obtain \( \{o'_1, \ldots, o'_m\} \sim \hat{f}_{h,K} \). Due to the fact that the new samples are not integers, we apply the floor function to them obtaining \( \lfloor \{o'_1, \ldots, o'_m\} \rfloor \).

2) Parameter estimation for RANDOM: In this case, we need to approximate \( P(o,d) \) since RANDOM depends on the number of objects and the average number of references. We apply the same idea as before, but to pairs \( (o,d) \) by using \( \hat{f}_{h,K} \) in 2 dimensions.

3) Parameter estimation in rEMF: As explained before, to generate models with rEMF, we need to define a set of rules \( \mathcal{G} \). Our approach is to manually define these rules for the relevant meta-model elements, and then we estimate the parameters of each rule individually. Since rEMF supports two types of rules, we need to estimate two types of parameters:

**Shape of distribution.** For normal rEMF rules, we are interested in determining which is the distribution that best represents the characteristics of each type of meta-model element in the set of realistic models. To do so, we estimate the parameters of each distribution associated to the rules. For each different rule, we build a set of samples. For instance, for rule 1 in Fig. 6, a sample is the count of the number of classifiers of a \( \text{E} \)Package. As a result, the set of samples \( \{c_1, \ldots, c_s\} \) is obtained. Then, for each available distribution in rEMF (Uniform, Normal, Negative Binomial and Poisson), we estimate its parameters using maximum likelihood. Finally we take the best distribution that fits \( \{c_1, \ldots, c_s\} \), using the log-likelihood as the comparison criteria.

**Priority in alternative rules.** In rule 2 of Fig. 6, the priorities of each one of the alternative rules (\( \text{Enum}, \text{DataType}, \text{Class} \)) need to be fixed. To do so, for each model we count the proportion of classifiers that it has (i.e. \( \text{EEnum}, \text{EDataType} \) and \( \text{EClass} \)). Then, we compute the mean with respect to all models. Finally, these averages are divided by the minimum average. Since priorities are integers, we round them.

### C. Applying GNNs to graph models

In the previous section we have explained how we estimate the parameters needed to force the generator to generate synthetic models that are as similar as possible to the dataset of real models. At this point, we have two sets of models (realistic models and synthetic models). We need a classifier to distinguish between these two sets in order to perform the C2ST. Therefore, we choose a GNN as our classifier. This neural network will receive a model (previously converted into a graph as described in Sect. II-A) as input and determine if it is real or not. Since we are dealing with graphs with labeled edges and nodes, we will consider the PyTorch [25], [26] implementation of the graph convolutional layer proposed in [27]. Therefore, the node embeddings are calculated using the following formulas:

\[
h^0_v = E_{\text{node}}(v) \quad (6)
\]

\[
h^r_v = \text{ReLU} \left( W^r_1 h^{r-1}_v + \sum_{r \in R} \sum_{w \in N_r(v)} \frac{1}{|N_r(v)|} W^r_1 h^{r-1}_w \right) \quad (7)
\]

where \( E_{\text{node}} \) is the embedding layer of node labels (initial embedding of the nodes), \( R \) is the set of references (edge types) and \( |N_r(v)| \) is the neighborhood of \( v \) restricted to \( r \). Note that equations 6 and 7 are adaptations of equations 1 and 2 respectively. In particular, the initial node embedding is given by an embedding layer and \( g_l \) is a function that takes into account the labels of the edges.

In order to have an embedding at the graph level, we will use an attention vector to summarize all the node embeddings i.e., the AGG operator in Eq. 3 is defined as follows:

\[
h^v_G = \sum_{v \in V} \alpha_v h^v_w \quad \text{where} \quad \alpha_v = \frac{\exp(\alpha \cdot h^L_w)}{\sum_{w \in V} \exp(\alpha \cdot h^L_w)} \quad (8)
\]

Finally, since we are interested in a binary classification problem, the output of the model is computed as

\[
P_{\text{model}}(\text{is real}?) = \sigma(W_F \cdot h^v_G + b_F)
\]
where $\sigma$ is the sigmoid function. Fig. 7 shows the attention mechanism and the last linear layer applied to the running example. There, once the graph convolutional layers are applied, each node $v_1, \ldots, v_n$ has an embedding associated $h_{v_1}^L, \ldots, h_{v_n}^L$. Each $h_{v_i}^L$ contains the information of the $L$-hoop neighborhood of $v_i$. To obtain the embedding of the entire graph, a weighted averaged is performed over the node embeddings using Eq. 8 (dotted lines that connect each $h_{v_i}^L$ with $\alpha$). The greater the weight associated to a node is, the more influence it has on the output. For example, if the attention weight associated to the node $v_2$ is 0.3 while the weights associated to the rest are 0.1, then the neural network focuses more in $v_2$ and its $L$-hoop neighborhood than in other nodes. Finally, $h_G$ is passed through a dense layer followed ($W_F$ and $b_F$) by a sigmoid function ($\sigma$) to output probabilities.

In our experiments, all dimensions of the layers and vectors are fixed to 64. We take $L$ = 2 since the diameter of the inputs graphs is relatively small. With this setting, the GNN takes into account the 2-‐hoop neighborhood. Therefore, all metrics studied in [7] are implicitly considered by our neural model. Finally, the neural network is trained using the binary cross-entropy loss and the Adam optimizer [28].

The training phase of the final model is depicted in Fig. 8. Each node of the graph is mapped to an initial vector by using the embedding layer $E_{node}$. These vectors are passed through a two layer GNN. After that, they are summarized into a vector that represents the entire graph by using the attention vector $\alpha$. The graph vector is the input of a fully connected layer ($W_F$, $b_F$) that outputs probabilities. Finally, the output is compared with the ground truth (using the binary cross-entropy loss) and all weights of the neural model are updated (i.e., weights of $E_{node}$, GNN$_1$, GNN$_2$, $\alpha$, $W_F$ and $b_F$) using backpropagation and the Adam optimizer.

D. Assessing realistic generators

Putting all together, a generator (gen) is a black box that receives as input a set of conditions (that can be a meta-model, some restrictions, etc) and outputs or samples a model $M$ (that is consistent with the input conditions, i.e., conforms to a meta-model, satisfy some restrictions, etc). As it was explained, this black box can be seen as a probability distribution over models $P_{gen}(M)$. We can suppose that all realistic models (models that people make) are sampled from a probability distribution $Q_{real}$. Thus, we can say that a generator is realistic if $P_{gen}$ is similar to $Q_{real}$.

In practice, we have a set of real models $\{R_1, \ldots, R_n\}$ sampled from $Q_{real}$ and a set of models $\{S_1, \ldots, S_n\}$ sampled from $P_{gen}$. To determine if $P_{gen} \approx Q_{real}$, we apply the following hypothesis test:

$$H_0: P_{gen} = Q_{real}$$

$$H_1: P_{gen} \neq Q_{real}$$

Using C2ST (Sect. II-D) and a classifier (the GNN model explained in Sect. III-C), we can compute the accuracy statistic and the $p$-value. If $p$-value $< \alpha$, then we reject $H_0$ and the generator is not realistic. The bigger $p$-value the better. Moreover, if the accuracy is $\sim 0.5$, the generator outputs realistic models. On the other hand, if the accuracy is high, it means that the generator is not realistic with respect to the dataset.

The application of this procedure to assess concrete model generators is explained in detail in the next section.

IV. EXPERIMENTS

In this section we report the results of our experiments applying our approach to assess whether state-of-the-art model generators are able to produce realistic models. We focus on the generators already explained in Sect. III-A that is, EMF random instantiator [18], RandomEMF [21], VIATRA [1] and Alloy [19], [20].

A. Domains

Our experiments were based on three case studies, each one corresponding to a different domain. The selection of the case studies was driven by the availability of public models. In particular, we obtained these models from the MAR search engine [16].

- **Ecore**. We considered a reduced version of Ecore [14], as depicted in Fig. 1 which included six well-formedness constraints (denoted $\Psi$ in Sect. III-A). Then, we downloaded 500 real models from MAR (which were crawled from GitHub) and transformed them into our reduced version. To ensure that the generators could generate similar models, we chose models that verify the following:

![http://mar-search.org](http://mar-search.org)
two conditions: the model must have just one EPackage as root and the model does not contain attributes of type EInt, EString, etc.

- **Yakindu Statecharts** [29] is an industrial modeling environment. We reused the meta-model and the ten well-formedness constraints defined in [5]. Regarding the dataset real models, a set of 369 has been considered (which were crawled from GitHub).

- **Database models** from GenMyModel. We considered a manually constructed version of its meta-model (depicted in Fig. 9) that includes tables, indices and references. Regarding well-formedness constraints, we manually derived three constraints according to our observations using the platform. Then, we built a dataset of 500 real models.

### B. Procedure

Given an initial set of real models \( R \) and a generator, each experiment is performed by following these steps, which are illustrated in Fig. 10.

1) \( R \) is split into \( R_1 \) (50%) and \( R_\text{II} \) (50%). This step corresponds to label 1 in Fig. 10. The goal is to ensure that the assessment procedure (C2ST) is not biased by the parameter estimation.

2) \( R_\text{II} \) is used to estimate some parameters of the generator (i.e., \( P(o) \) in VIATRA and Alloy, \( P(o,d) \) in RANDOM, shapes and priorities of \( G \) in rEMF). This step corresponds to label 2 in Fig. 10.

3) Once the parameters of the generator are fixed, the generator is used to generate \( n = |R_1| \) models, which form the set of synthetic models \( S \) (label 3 in Fig. 10).

4) The sets \( R_0 \) and \( S \) are merged and shuffled (label 4).

5) Finally, we apply C2ST considering that \( S \sim P_{\text{gen}} \) and \( R_1 \sim Q_{\text{real}} \) (label 5). To do so, the merged set of models is split into training/validation/test (60%/15%/25%), label 5.1. The training set is used to train the GNN model and the validation set is used to perform early stopping (label 5.2). As a result, a trained GNN model is obtained. This model is evaluated using the test set (label 5.3). Finally, the accuracy and \( p \)-value are calculated using equations 3 and 5 respectively. The results are used to determine whether the generator is realistic or not.

### C. Results

The results of the C2ST for each generator and for each domain are shown in Table I. According to them, we can draw the following conclusions:

- No generator can be considered realistic since all \( p \)-values are less than \( \alpha = 0.01 \). Therefore, we cannot accept \( H_0 : P_{\text{gen}} = Q_{\text{real}} \).

- The least realistic generator is RANDOM. This is caused by the fact that it actually generates models that conform to a meta-model, but the majority of the generated models are inconsistent with the well-formedness constraints. This is because we did not apply OCL validators to rule out invalid models before applying the classifier, for two reasons. First, to follow the same procedure used in related works [8], and secondly because RANDOM was only able to generate valid models when they are small. For example, in the Ecore case study, 95% of the generated models are invalid (violate at least one constraint) and the remaining 5% models are small and simple. This fact also explains the accuracy value of 1: the classifier detects the constraint violations or simple models with small size.

- The most realistic generator is rEMF. This could be justified for two reasons. First, rEMF is the most customizable generator since the user has to define the set of rules \( G \) that guides the model generation. And second, there is more observation of the set \( R_\text{II} \) since we estimate the parameters of many distributions (for example, in the Ecore domain, number of classifiers per EPackage, number of features per EClass, etc) to construct \( G \). However, in RANDOM, Alloy and VIATRA only \( P(o,d) \) and \( P(o) \) are set.

Altogether, these experiments show that the distributions of realistic models have a complexity which cannot be ap-
proximated by the distributions implemented by current model
generators. Therefore, new techniques and procedures are
needed to enable the faithful generation of realistic models.

D. Interpretation

The described procedure tells us whether a generator is
realistic or not. However, faced with a negative answer we
need some mechanism to interpret this result and understand
it. Since our procedure is based on C2ST, we can determine
which models of the test set have been correctly or incorrectly
classified by the GNN. Then, for each interesting model, we
can inspect the behavior of the neural network when this
particular model is introduced as input. Our GNN model has
an attention mechanism that can be easily interpreted by just
looking at its weights. This feature is particularly useful when
the input model is large. In this case, we can focus only on
the parts of the model that the GNN has focused on.

As a concrete example, let us consider the worst and best
generators: EMF random instantiator and RandomEMF. For
each generator we show one example model per domain which is
not considered realistic by the corresponding GNN. We use
a custom notation in which each object in the model is given a
color according to the attention that the GNN has put on it (see Fig. 11 and Fig. 12). These graphics are automatically
generated using specific tools that we have developed.

EMF random instantiator is not considered realistic because
its generated models violate the well-formedness constraints.
In almost all the analyzed models, the GNN focuses on some
object that violate some constraint. For instance, Fig. 11 shows three examples of invalid models. In the Ecore model
(label 1.), the GNN focuses on a EClass that inherits from
itself. In the Yakindu domain (label 2.), the model focuses
on transitions whose target state is an entry (this violates a
constraint). Finally, label 3. shows a synthetic database model,
for which the GNN focuses on references whose target and
source columns are the same column (this again violates a
constraint).

Regarding RandomEMF, the criteria followed by the GNN
to distinguish between generated and real models is more
diverse (Fig. 12). For instance, in the case of Ecore (label 4.),
the GNN focuses on EEnum objects that are not referenced by
any EAttribute belonging to an EClass of the same package.
The GNN correctly detects that it is an uncommon situation
to create an EEnum but not using it in the meta-model. In
Yakindu models (label 5.), the GNN identifies states that are
isolated (i.e., they are not pointed to any transition). This is
a common pattern in models generated by RandomEMF but not in real models. In database models (label 6.), we can
observe that there are cases in which RandomEMF generates
two objects Reference that are opposites i.e., the target column
of one of them is the source column and vice versa. This
pattern is allowed (you can instantiate it by using the editor
and it does not break any constraint) but it does not make
sense and it is not common in real models.

Data Availability. The replication package together with all
attention heatmaps of the test models for each generator and
for each domain are available at https://github.com/Antolin1/
TCRMG-GNN [30].
TABLE I
RESULTS OF C2ST.

<table>
<thead>
<tr>
<th>DOMAIN</th>
<th>RANDOM</th>
<th></th>
<th>ALLOY</th>
<th></th>
<th>VIATRA</th>
<th></th>
<th>REMF</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>accuracy</td>
<td>p−value</td>
<td>accuracy</td>
<td>p−value</td>
<td>accuracy</td>
<td>p−value</td>
<td>accuracy</td>
<td>p−value</td>
</tr>
<tr>
<td>Ecore (</td>
<td>Dtest</td>
<td>= 125)</td>
<td>1</td>
<td>2.544E-29</td>
<td>0.9600</td>
<td>4.078E-25</td>
<td>0.9280</td>
<td>5.326E-22</td>
</tr>
<tr>
<td>Yakindu (</td>
<td>Dtest</td>
<td>= 93)</td>
<td>1</td>
<td>2.614E-22</td>
<td>0.9462</td>
<td>3.759E-18</td>
<td>0.9569</td>
<td>6.033E-19</td>
</tr>
<tr>
<td>Database (</td>
<td>Dtest</td>
<td>= 125)</td>
<td>0.9920</td>
<td>1.880E-28</td>
<td>0.9200</td>
<td>2.957E-21</td>
<td>0.9680</td>
<td>6.267E-26</td>
</tr>
</tbody>
</table>

Fig. 11. Attention heatmap of models generated with EMF random instantiator

Fig. 12. Attention heatmap of models generated with RandomEMF

- Moreover, to the best of our knowledge, this is the first work that applies GNN to software models. Thus, our proposed GNN architecture can be adapted to face other model classification problems (e.g., meta-model classification [11], [12], UML classification [13], etc).

This paper presents a novel procedure to assess realistic
model generators. The approach is interpretable since it is based on the transparent test C2ST and uses a GNN with an attention mechanism.

VI. RELATED WORK

In this section we review works related to our proposal, organized in three categories: two-sample tests, assessment of realistic generators and model generators.

A. Two-sample tests

The aim of these tests is to determine if two sets of samples come from the same distribution. Traditional examples are Student t−test and Kolmogorov-Smirnov (KS) test. Although these tests are widely used, they require strong assumptions about the data (e.g., normality assumption) [33]. Furthermore, they cannot be applied to discrete data (such as graphs or strings). In this line, the Kernel Two-Sample Test [34] is a powerful test based on kernels and the Maximum Mean Discrepancy (MMD). It can be applied to any type of data as long as you can define a kernel function. On the other hand, another popular test is the Classifier Two-Sample Test [10]. The idea is that under the assumption of the null hypothesis, the classification problem of distinguishing between samples from a distribution and the other is impossible. It can be proved that this test is a particular case of the Kernel Two-Sample Test [35]. We use the Classifier Two-Sample Test in our approach.

B. Assessment of realistic model generators

In [7], the authors propose the use of multidisciplinary graph metrics to characterize realistic models. The work in [5] uses these metrics to compare two generators. Given a graph statistic (degree distribution, multiplex participation coefficient, pairwise multiplexity, etc), they measure how realistic a generator is by using the average value of the KS statistic between each pair of models \((A, B)\) where \(A\) belongs to the set of real models and \(B\) to the set of generated models. This technique has three important shortcomings. Firstly, summarizing an entire graph into a set of graph metrics causes an information loss. Secondly, a subset of graph statistics have to be chosen to perform the assessment, but not all metrics are equally effective to perform this task [7]. Thirdly, this approach is not interpretable. Moreover, the KS based measure is not consistent since it does not take into account the differences inside the set of real models and inside the set of synthetic models. On the other hand, [8] claims that a generated model is realistic if its distance (considering graph metrics and statistics) with respect to its nearest neighbor real model is less than a threshold. To measure how realistic a generator is, the authors propose to take the average of all the distances and the percentage of realistic synthetic models. This proposed method is not consistent since it can be hacked by a generator that always generates the same realistic model. In our approach, the GNN learns the structure of realistic models by looking for differences between both synthetic and realistic sets of models, so it is robust to this scenario. An evaluation metric based on the Maximum Mean Discrepancy [34] is proposed in [35]. This metric is used to assess generative models of graphs and it is similar to the one proposed in [5] (except that it takes into account the differences inside the groups which makes it a consistent measure).

C. Model generators

Some model generators are based on mapping the meta-model and constraints to logical formulas and use SAT-solvers to obtain consistent models. It possible to build this type of generators using model finders like Formula Framework [36], Alloy [20] and EMF2CSP [37]. In rule-based generators, the generation process is guided by rules. Examples of this type of generators are RandomEMF [21], [38] or [39]. Search-based generators transform the generation into a search problem. Examples are VIATRA [1], [40] and [41]. Random models of graphs (such as Erdős-Rényi [42] or Watts-Strogatz [43]) can be considered model generators. However, they are mostly for unlabeled graphs. Since there exist networks whose complexity is beyond these random models, some generative models of graphs have been proposed. For instance, GraphRNN [35] uses a two-level recurrent neural network to generate a graph. GraphVAE [44] uses variational autoencoders. [45] transforms the generation of a graph into a sequence of actions and uses a GNN to decide which action should be taken in each step. These generative models need a training set and they are able to generate graphs whose properties are close to the training set. They can be used in model generation because they can be adapted to generate labeled graphs.

VII. CONCLUSION

In this work we have presented a novel method to assess whether a given generator is realistic or not. Our approach is based on training a GNN in the task of distinguishing between real models and synthetic models. If the classifier does not achieve a good performance, the generator is realistic. The proposed method is interpretable and some hints of why the generator is not realistic can be derived by looking at the samples in the test set and the attention weights of the GNN model. On the other hand, the results of applying the proposed assessment to four state-of-the-art generators in three different domains show that none of the generators is realistic. Therefore, more research is needed to obtain realistic model generators.

As future work, we will try to automate the interpretation of our approach. Furthermore, we plan to include this approach in a generator in order to guide the generation process and obtain realistic models.

ACKNOWLEDGEMENTS

We would like to thank Kristóf Marussy for his guidance in the use of VIATRA generator in the experiments section.

Work funded by a Ramón y Cajal 2017 grant (RYC-2017-237) funded by MINECO (Spain) and co-funded by the European Social Fund. José Antonio Hernández López enjoys a FPU grant funded by the Universidad de Murcia.