Generating structurally realistic models with deep autoregressive networks

José Antonio Hernández López and Jesús Sánchez Cuadrado

Abstract—Model generators are important tools in model-based systems engineering to automate the creation of software models for tasks like testing and benchmarking. Previous works have established four properties that a generator should satisfy: consistency, diversity, scalability, and structural realism. Although several generators have been proposed, none of them is focused on realism. As a result, automatically generated models are typically simple and appear synthetic. This work proposes a new architecture for model generators which is specifically designed to be structurally realistic. Given a dataset consisting of several models deemed as real models, this type of generators is able to produce new models which are structurally similar to the models in the dataset, but are fundamentally novel models. Our implementation, named MODELMIME (M2), is based on a deep autoregressive model which combines a Graph Neural Network with a Recurrent Neural Network. We decompose each model into a sequence of edit operations, and the neural network is trained in the task of predicting the next edit operation given a partial model. At inference time, the system produces new models by sampling edit operations and iteratively completing the model. We have evaluated M2 with respect to three state-of-the-art generators, showing that 1) our generator outperforms the others in terms of the structurally realistic property 2) the models generated by M2 are most of the time consistent, 3) the diversity of the generated models is at least the same as the real ones and, 4) the generation process is scalable once the generator is trained.

Index Terms—Model generators, Realistic models, Graph neural networks, Recurrent neural networks, Generative models.

1 INTRODUCTION

THE automated generation of models is a key approach in many areas of software and system engineering such as testing and benchmarking of graph databases [1], [2], creation of complex test stubs in the object-oriented field [1], [3] or the synthesis of prototypical test contexts in the assurance of smart cyber-physical systems [1], [4], [5].

The tools that automatically generate models are called *model generators*. A model generator receives as input a set of specifications describing how the output models should be, and it is in charge of producing models that satisfy these constraints as much as possible. Typically, the inputs include the meta-model that the output model will conform to, the size of the output models, OCL or graph constraints, etc. Several generators have been proposed in the last years, which can be classified as SAT-solver based generators [6], [7], [8], [9], generators based on constructive formalisms like grammars [10], [11], search-based model generators [1], [12], [13], [14] and purely random generators [15].

In this context, Varró et al. [5] established four properties that a generator should satisfy: 1) the generator must be *consistent* to satisfy a set of domain constraints (e.g. specified with OCL [16] constraints or graph patterns [17]); 2) the generator must be *diverse*, that is, the generated models should contain a wide variety of shapes [18]; 3) the generator should be *scalable* with respect to the size of the output model and 4) the generator must be *realistic*, that is, the generated models cannot be distinguished from real ones made by humans. 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 (i.e., ignoring the attribute values). *Consistency* and *diversity* are desirable in a functional testing scenario, whereas *scalability* and *realism* are essential for benchmarking and stress testing [1]. For example, *realism* is an important property for testing in domains like autonomous cars [12], in which test models are intended to represent real-life scenarios [19], [20], but unrealistic test cases are less useful because they may not happen in the real world. Thus, realistic test scenarios are more faithful to reality and they are preferred when testing this type of systems. Moreover, test beds might not be shared due to intellectual property issues and the ability to automatically generate structurally similar models can be a key feature to facilitate testing tasks.

A limitation of current model generators is that none of them is focused on making the generated models look real. This means that the generated models are typically simple and synthetic and not similar to the ones made by humans. On the other hand, satisfying the four properties at once is a challenge. For instance, SAT-solver based generators like Alloy [6], [7] or Formula [9] are consistent but they are neither diverse [18], scalable [1] nor realistic [21]. The VIATRA generator [1] is consistent, diverse and scalable but it is not realistic [22].

This work proposes a new breed of model generators that is focused on satisfying the realistic property. Given a dataset consisting of several models deemed as *real models*, this type of generators is able to produce new models which share similar characteristics, but are fundamentally novel models. The key idea is that models can be broken down into a sequence of edit operations. Thus, we can train a

• *J.A. Hernández López and J. Sánchez Cuadrado are with the Department of Computer Science and Systems, Universidad de Murcia, Spain. E-mail: {joseantonio.hernandez6, jesusc}@um.es*

deep autoregressive model [23], [24] to predict the next edit operation using as input a partial model [25]. Then, we use the trained model to incrementally build fresh models by iteratively inferring the next edit operation. To this end, we have designed an architecture based on combining a Graph Neural Network (GNN), for selecting edit operations, and a Recurrent Neural Network for selecting the nodes for the selected edit operation. We have evaluated our implementation with respect to the structurally realistic property, using the Maximum Mean Discrepancy [26], [27] (a very well-known metric used to measure the quality of generative models) over the graph metrics proposed in [21]. We show that our generator outperforms three state of the art generators in this property. Regarding the other three properties (consistency, diversity and scalability), we show that more than the 85% of generated models are consistent, the diversity of the generated models is at least the same as the real ones and the generation process is scalable once the generator is trained.

Altogether, the contributions of this work are the following:

1) The use of the notion of *edit operation* to decompose a model into a sequence of operations which represent the structure of the model and can be learned. This allows us to transform the model generation problem into sequence generation that can be tackled by a generative model.
2) A generator, named ModelMime (M2), that is focused on producing realistic models that are structurally similar to a given dataset. To the best of our knowledge, this is the first model generator that is specifically designed to satisfy this property.
3) We replicate the metrics for consistency and diversity proposed by previous works [5], [18] to compare our generator with three state-of-the-art generators. Moreover, we propose the use of the Maximum Mean Discrepancy (MMD) metric to assess the realism of each model generator.

Organization. Section 2 gives a brief explanation of the technical background that underlies our approach. Section 3 presents the neural architecture that we have designed. Section 4 describes the experiments that we have carried out to compare our generator with three state-of-the-art generators. Finally, Section 5 discusses the related work and Sect. 6 concludes.

2 Background

This section presents the background of our approach. As a running example, let us suppose that we want to generate Yakindu Statecharts [28], which is a widely used formalism in model-based engineering. The meta-model is shown in Fig. 1a and Fig. 1b shows an example model. The example has 8 objects: a *Statechart* with one *Region* depicted as a rectangle, an *Entry* (black circle) to represent the initial state, two *States* represented as rounded rectangles and three *Transitions*.

2.1 Models as graphs

Models can be seen as graphs [22], [29] since a one-to-one function between models and graphs can be constructed by considering the multigraph of their abstract syntaxes. Thus, given a model that conforms to a meta-model, one can construct a labeled multigraph (ignoring attribute values in this case) by doing the following:

- Each object of the model is mapped to a vertex and it is labeled with the name of the meta-class that the object belongs to.
- Each reference in the model is associated to an edge, which is labeled with the name of the reference.

For instance, when we apply this procedure to the model in Fig. 1b, the graph in Fig. 2 is obtained. It is important to note that, we do not consider attributes since we are focused in the *structurally realistic* property (see Sect. 2.2.1).

2.2 Model generators

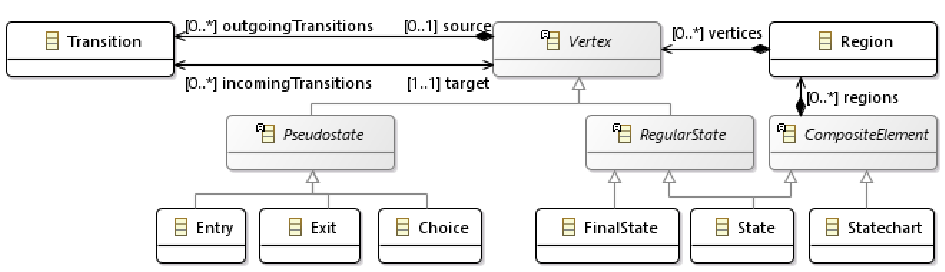
A model generator is an artifact that receives a set of conditions $\{c_1, \ldots, c_n\}$ and attempts to output models that satisfy these conditions. We can see generators as implicitly defining a conditional distribution over models $P(M|c_1, \ldots, c_n)$ since they are not deterministic. That is, once the input conditions $\{c_1, \ldots, c_n\}$ are established, we can run the generator and sample a model M. For example, let us consider the VIATRA generator [1]. This generator receives as input conditions a meta-model $\mathcal{M}$, a set of constraints Ψ and the number of objects o that the output model will have. According to the previous definition, it defines a distribution $P_{\text{VIATRA}}(M|o, \mathcal{M}, \Psi)$.

2.2.1 *Properties of model generators*

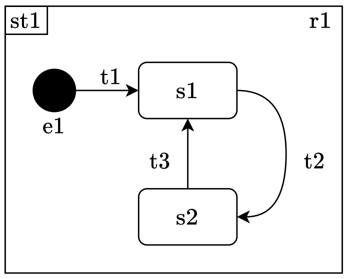
Varró et al. [5] presented the Graph Model Generation Challenge in which four properties that a generator should satisfy are established, namely:

- **Consistency**: A generator is consistent if 1) the generated models conform to a given meta-model $\mathcal{M}$ and 2) respect a given set of domain constraints Ψ. For example, for our running example (Fig. 1) we define two constraints (extracted from [5]):
  1) There is no *Transition* whose target state is an *Entry*.
  2) A *Region* must contain one and only one *Entry*.
- **Diversity**: A generator is diverse if the generated models contain a wide variety of shapes. The diversity for a concrete model is measured by using the *internal diversity* [18].
- **Scalability**: A generator is scalable if it is able to generate models with a significant amount of objects within a reasonable period of time.
- **Realism**: A generator is realistic if the generated models cannot be distinguished from real ones. In particular, a generator is structurally realistic if the set of generated models cannot be distinguished from the real ones by just looking at the typed graph structure (i.e., ignoring the attribute values) [5], [12].

Our generator is focused on the structurally realistic property which is a challenge itself [12]. For example, Fig. 3a shows one real Yakindu model extracted from GitHub. Fig. 3b depicts a model generated by the VIATRA generator which cannot be considered realistic when compared to real ones. Finally, Fig. 3c shows a model generated by M2 which does ressemble real ones. In the rest of the paper we describe



(a) Excerpt of the Yakindu meta-model, extracted from [5]



(b) Example model

Fig. 1: The Yakindu Statecharts meta-model and one example model

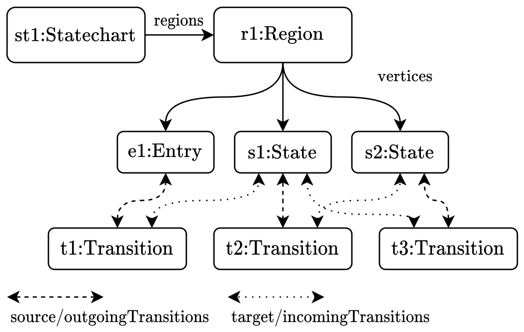
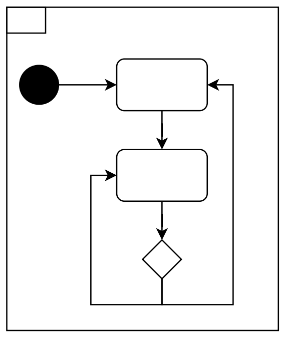
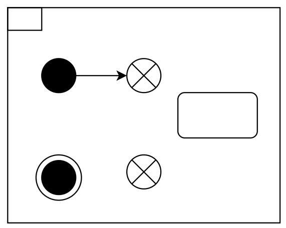


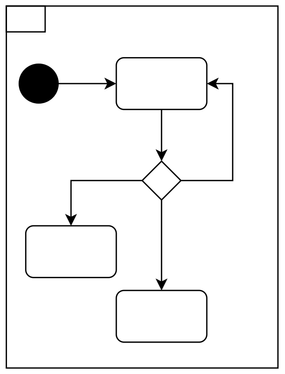
Fig. 2: Multigraph associated to the model in Fig. 1b. To avoid too many edges, the opposite references (*source* / *outgoingTransitions* and *target* / *incomingTransitions*) are represented by arrows with two heads. In particular, *source* / *outgoingTransitions* is represented by a dashed arrow and *target* / *incomingTransitions* by a dotted one.



(a) Model extracted from GitHub



(b) Model generated using VIATRA generator



(c) Model generated using M2

Fig. 3: Example of real and synthetic models.

our technique to achieve this result. On the other hand, in these examples, the models do not have concrete values for the attributes. Building a fully realistic generator also involves dealing with attribute values, but to tackle this task we need, first of all, to have models with a realistic structure.

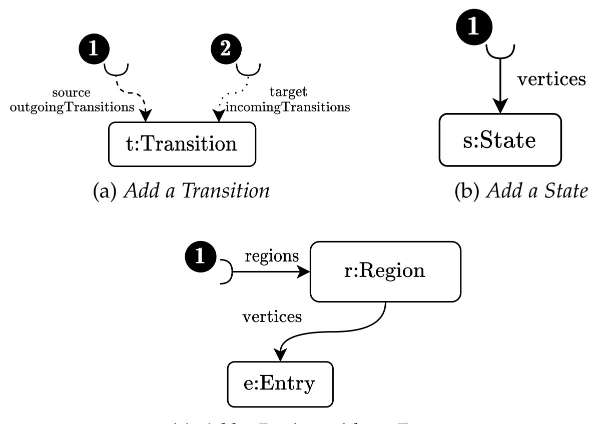
2.3 Models as a sequence of edit operations

In our approach, we use a sequential generative model [24], which is based on adding nodes and edges sequentially as a way to synthesize models. This means that we need model construction operations. To this end, we use the idea of *edit operation* [25]. An edit operation is an abstraction that represents an operation that a modeler performs when she is building a model. For our case, we focus only on operations that add elements to the model which we define with three components:

- A unique **identifier** to refer to the type of the edit operation univocally.
- An **edit graph**. It represents graph of objects that will be added to the model under construction.
- A set of **edit connections**. They are the edges that will connect the edit graph with the model under construction. Each edit connection requires the selection of a node in the model under construction to link the edit graph with the model.

Let us consider the meta-model of our running example (Fig. 1b) and let us define a set of edit operations composed by *add a Transition*, *add a State* and *add a Region with an Entry*. Fig. 4 shows these three edit operations. The arrows ended by a connector (i.e., a semi-circle ⊂) are the edit connections and the rest of the graph is the edit graph. In the case of *add a Transition* the edit graph is composed by a node and the edit connections are the references *source* / *outgoingTransitions* and *target* / *incomingTransitions*. The edit graph of *add a State* is also composed by a single node and the edit connection is the reference *vertices*. Finally, the edit graph of *add a Region with an Entry* is composed by two nodes (a *Region* and and *Entry*) connected by the reference *vertices* whereas, the edit connection is the reference *regions*.

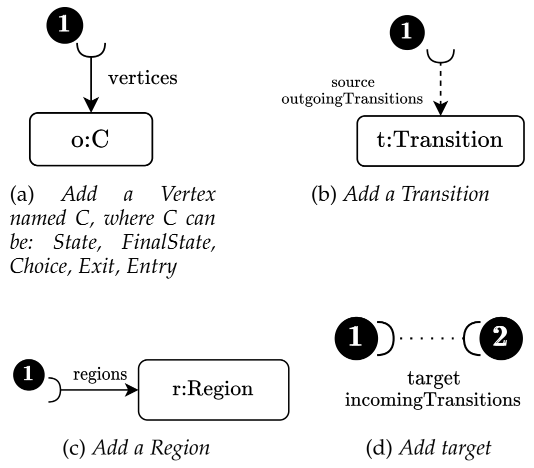
The application of an edit operation is carried out by selecting the nodes in the model under construction that will be connected to the edit graph through the edit connections.



(a) *Add a Transition* (b) *Add a State*

(c) *Add a Region with an Entry*

Fig. 4: Illustrative edit operations for the running example.



(a) *Add a Vertex named C, where C can be: State, FinalState, Choice, Exit, Entry* (b) *Add a Transition*

(c) *Add a Region* (d) *Add target*

Fig. 5: All atomic operations extracted from the Yakindu meta-model in Fig. 1a.

The number of nodes to be selected is equal to the number of connectors (⊂). For example, Fig. 6 shows how *add a Transition* is applied:

1) Select the first connector ❶ i.e., the source of the *Transition* (*Entry e1*).
2) Select second connector ❷ i.e., the target of the *Transition* (*State s1*).
3) Add the edit graph to the model under construction i.e., a transition that connects the selected nodes.

Therefore, in our approach we iteratively construct models by selecting and applying edit operations as explained.

2.4 Definining model operations

To train a model generator for a given meta-model with its constraints, we need to define the corresponding edit operations. There are two types of edit operations:

- *Atomic edit operations*. An atomic operation corresponds to the minimal set of changes needed to extend the current model with a new object or a new reference between two objects. Fig. 5 shows the atomic operations derived from the meta-model in Fig. 1a. There are eight atomic edit operations: one for each type of *Vertex*, *add a Transtion*, *add a Region* and *add target*. These atomic operation can be automatically derived from the meta-model by traversing all references. If a reference is *containment* then an edit operation of type "add object" is defined for each potential concrete class referenced (similar to the *additional node creation* rule of [30]). For instance, Fig. 5a shows that from the *vertices* reference five different edit operations are derived. This type of atomic operation, not only creates the object, but it also places the object in its containment reference. On the other hand, if a reference is *non-containment*, an operation like Fig. 5d is derived (similar to the *additional edge creation* rule of [30]).
- Complex edit operations. These operations cannot be directly extracted from the meta-model and have to be defined manually. They are defined by combining several atomic edit operations. For instance, the edit operations presented in Fig. 4a (obtained by combining the atomic edit operations of Figs. 5b 5d) and Fig. 4c (combination of Figs. 5a and 5c) are complex edit operations. This type of edit operations helps our generator to produce more consistent models and to improve its scalability.

Therefore, from the perspective of a user who wants to use M2 with a specific meta-model, the definition of the edit operations is a pre-requisite. First, all possible atomic edit operations must be defined to ensure that the generator can "cover" the complete meta-model. This can be done automatically [30]. At this point it is possible to directly use M2, but it is advisable to use domain knowledge to define several complex edit operations to help M2 produce better models. For instance, the edit operation depicted in Fig. 4c ensures that all *Regions* will have one *Entry*, which is a domain constraint. Thus, there are two possible configurations to run our generator (Sect. 3): 1) only atomic edit operations (automatically generated) and 2) atomic edit operations plus user-defined complex edit operations. This set of edit operations is used by M2 both in the training and inference phase.

3 Approach

Our generator relies on a neural model which guides the generation process. Fig. 7 shows a high-level view. This neural model is trained to produce models that are similar to an input dataset. Its architecture combines a Graph Neural Network (GNN) and a Gated Recurrent Unit (GRU). The GNN [31] is used for selecting edit operations (i.e., the identifier of the edit operation), whereas the GRU [32] is needed for selecting which nodes in the current model are connected to an edit operation.

Our approach is composed of two phases that involve the neural model:

1) **Training phase**. The generator receives the definition of the edit operations associated to the meta-model and a dataset of models conforming to such meta-model. Each model of the dataset is broken down into a sequence of edit operations and the neural network is trained to predict the next edit operation given a partial model.

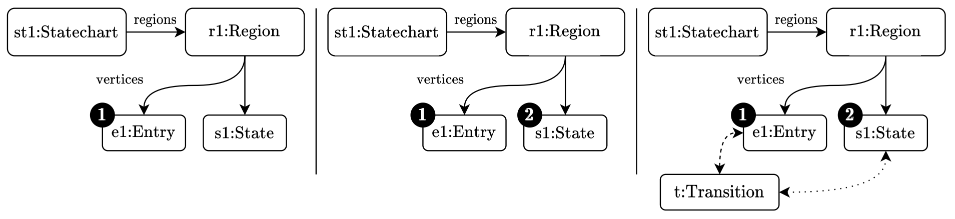


Fig. 6: Application of *add a Transition* to a model under construction. Step 1: select the node for the *source* reference. Step 2: select the node for the *target* reference. Step 3: Add the edit graph with the connecting nodes.

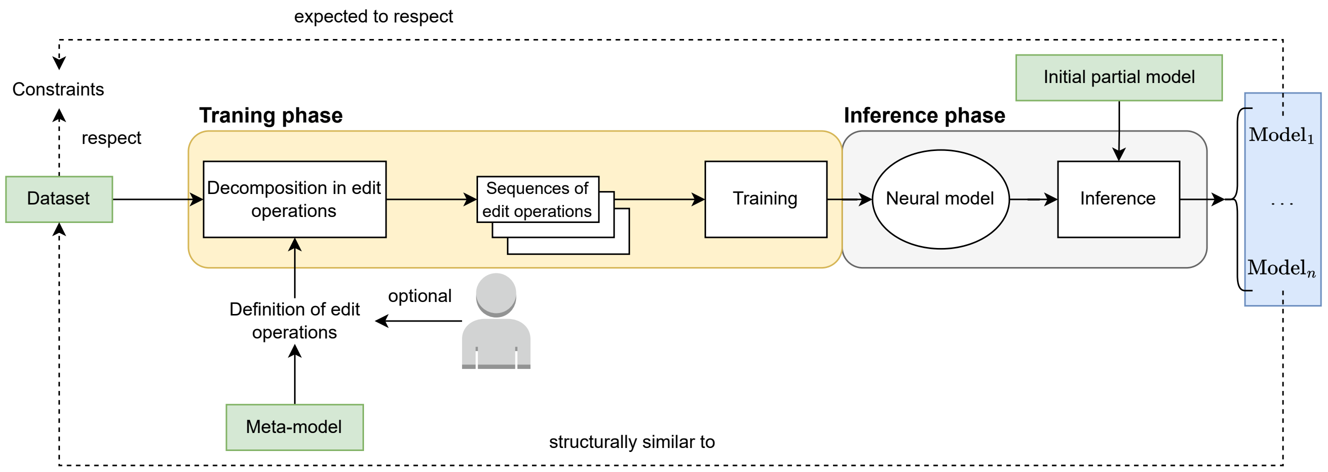


Fig. 7: Proposed generator.

2) **Inference phase**. Once the network is trained, it takes an initial partial model as input and samples an edit operation. Then, this edit operation is applied to grow the model. The process is repeated sequentially until the generator orders to stop or the size of generated model reaches a given threshold. The provided initial model can be empty. In this case, our generator will use as seed model a root object (e.g., an object of type *Statechart* in our running example). Also it is possible to input a more complex model that establishes the minimal structure of the generated models.

To use our generator the user has to provide a meta-model and a dataset of models conforming to this meta-model. The atomic edit operations can be derived from the meta-model and the user can optionally provide more complex edit operations. The neural model will produce new models that will be similar in structure to the dataset by applying edit operations in sequence. Precisely, which edit operation to apply next is what the neural model learns. It is important to note that our neural model does not receive domain constraints as input. Instead, the models in the training dataset must respect the constraints. The goal is that the generator learns the structure of the training models and thus avoids creating models which do not satisfy the constraints. In other words, as the models of the dataset respect a set of constraints, we expect that the generated models will respect them as well.

In the following, we will describe in detail the architecture of the proposed neural model, the inference phase and the training procedure.

3.1 Neural model

Given a dataset of real models $\{M_1, \ldots, M_n\}$, we assume that they come from a distribution over models Q_{real}. Thus, we propose to learn a distribution $P_\theta(M) = P(M|\theta)$ that is close to the real distribution Q_{real}. Here θ represents the parameters (weights) of the neural network.

The neural model tackles the model generation as a sequential process. In each step, a partial model is extended by applying an edit operation:

$$M^1, \ldots, M^T = M \text{ with } M^{i+1} = e_i(M^i) \text{ for all } i = 1, \ldots, T-1$$

where e_i is an edit operation. Therefore, our neural model is used to sample an edit operation $e_i \sim P_\theta(e_i|M^i, \ldots, M^1)$, and then e_i is applied to M^i to get M^{i+1}. The sequence of pairs given by a partial model and applied edit operation,

$$r = ((M^1, e_1), \ldots, (M^{T-1}, e_{T-1}))$$

is called *decoding route* [33]. It contains all the information required to reconstruct a complete model.

We will accept the Markov assumption. Therefore, this equality holds: $P_\theta(e_i|M^i, \ldots, M^1) = P_\theta(e_i|M^i)$ and our neural model receives the last partial graph as input. To sample an edit operation e_i from $P_\theta(e_i|M^i)$, the network performs the following steps:

1) **Edit operation selection** (f_{edit}), that is, select an edit operation from the set of defined edit operations. It is composed by two subtasks:
   a) **Identifier selection** (f_{id}), that is, select the edit operation identifier that will be applied.

b) **Connectors completion** (f^j_{conn}). We have to select the nodes in the model under construction that will be connected to the edit graph associated to the selected edit operation.

2) **Termination** (f_{end}). It establishes if we have to end the generation process.

Each step requires its own neural architecture, which are explained next.

3.1.1 Architecture for identifier selection

This neural module (f_{id}) is in charge of the inference of an edit operation. Fig. 8 shows the architecture. Given a partial model M^i, each of the nodes are passed through an embedding layer (E_{node}) to obtain the initial embeddings (❶ in Fig. 8). Then, a GNN of L layers is applied to these embeddings to obtain a contextualized embedding for each node, that is, an embedding that has information about its graph neighborhoods according to the input model (label ❷ shows that the GNN transforms the initial embeddings for each node into the contextualized embeddings). We consider the GNN layer proposed in [34] since it takes into account the node and edge labels. Thus, in this neural module, the node embeddings of each node in each layer l are computed as follows:

$$h^0_v = E_{\text{node}}(v) \in \mathbb{R}^d$$

$$h^l_v = \text{ReLU}\left(W^1_l h^{l-1}_v + \sum_{r\in\mathcal{R}} \sum_{w\in\mathcal{N}_r(v)} \frac{1}{|\mathcal{N}_r(v)|} W^r_l h^{l-1}_w\right) \in \mathbb{R}^d$$

where E_{node} is the embedding layer of nodes (initial embedding), $\mathcal{R}$ is the set of reference types defined in the meta-model (edge types when seen as a graph) and $\mathcal{N}_r(v)$ is the neighborhood of v restricted to r hops. Since we apply L layers, we denote the final node embedding matrix as $X = H^L \in \mathbb{M}_{n\times d}$ where $n = |V|$ and d is the hidden dimension. To represent the full input graph as a vector we use the average of all node embeddings as aggregation operation (label ❸ in Fig. 8):

$$h_{M^i} = \text{AVG}(X) \in \mathbb{R}^d$$

Finally, we pass this vector through two linear layers and apply a softmax activation function in order to assign a probability to each operation identifier (label ❹):

$$f_{\text{id}}(M^i) = \text{SOFTMAX}(W^2_{\text{edit}} \cdot \text{ReLU}(W^1_{\text{edit}} \cdot h_{M^i}))$$

Given a partial model, we use it as input to $f_{\text{id}}(M^i)$ to sample the identifier of the next edit operation according to the output probabilities. In this example, *add a Transition* is the most likely edit operation.

3.1.2 Architecture for connectors completion

This neural module is in charge of choosing already existing nodes in order to assign them to the connectors of the edit operation sampled in the previous step. In the example, the *add a Transition* operation needs to select two objects for the *source* and *target* references. The architecture to perform this task is depicted in Fig. 9 and an it relies on a GRU [32]. The rationale behind choosing a GRU as the core of this module is the fact that the set of defined edit operations could have different number connectors.

To be able to compute the node for the first connector, the GRU needs two pieces of information: the identifier of the selected edit operation and a vector representing the current partial model. We obtain this information from the previous module, which is h_{M^i} (label ⓐ in Fig. 8, the vector summarizing the model) and *add a Transition* (label ⓑ in Fig. 8, the selected edit operation). Then, if the edit operation has two or more connectors, we use the node embedding associated to the node selected by the GRU (which is computed by the f_{id} module, e.g., h^L_{e1}) as the next input for the GRU.

More formally, the recurrent network is initialized (label ❶) with the vector associated with the input model (h_{M^i}) and the sampled edit operation which is passed through an embedding layer. To sample the node for the first connector, we apply the GRU just one step, as follows:

$$h^{\text{GRU}}_0 = h_{M^i}$$

$$x_0 = E_{\text{id}}(\text{id}_{e_i})$$

$$h^{\text{GRU}}_1 = \text{GRU}(h^{\text{GRU}}_0, x_0)$$

To obtain the concrete node, we concatenate the output hidden state to each node embedding of the partial model (label ❷). We pass this concatenation through two linear layers and we apply a softmax (label ❸) to get the probability for each node of being selected for the connector. Formally, f^1_{conn} computes the probability of the first node as follows:

$$f^1_{\text{conn}}(M^i) = \text{SOFTMAX}\left(W^2_{\text{con}}\text{ReLU}\left(W^1_{\text{con}} \cdot \text{CONCAT}(X, h^{\text{GRU}}_1)\right)\right)$$

When the edit operation needs to be completed with more nodes (i.e., has two or more connectors) we use as input the embedding of the previously selected node, that is, when $j > 1$ (j represents the connector we are filling) we have the following (label ❹):

$$x_j = \text{Embedding of the node sampled with } f^{j-1}_{\text{conn}}.$$

$$h^{\text{GRU}}_j = \text{GRU}(h^{\text{GRU}}_{j-1}, x_j)$$

$$f^j_{\text{conn}}(M^i) = \text{SOFTMAX}\left(W^2_{\text{con}}\text{ReLU}\left(W^1_{\text{con}} \cdot \text{CONCAT}(X, h^{\text{GRU}}_j)\right)\right)$$

In this example, the node sampled for the *source* connector of the Transition is *e1*, the node sampled for the *target* connector is *s1*.

In this way, the task of applying an edit operation, f_{edit}, is defined as the concatenation of f_{id} and f^j_{conn}. Using the edit graph associated with the selected edit operation and the selected nodes, we update the current partial model. In this example, we have completed the current model with a new Transition node (the edit graph of the selected operation *add a Transition*) with connections to *Entry e1* and the *State s1*.

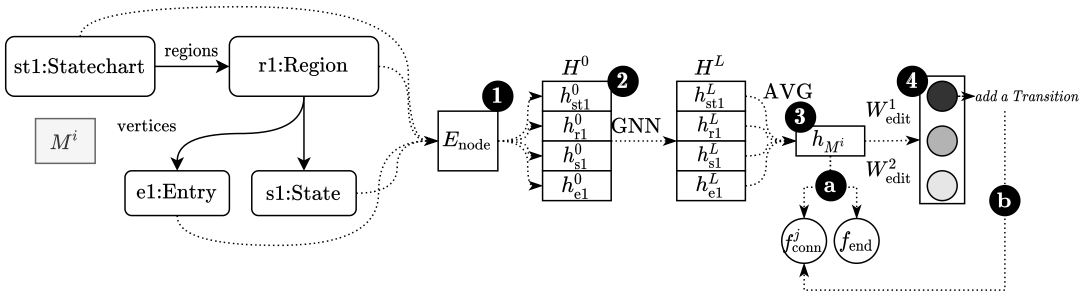


Fig. 8: Architecture of f_{id}. (1) embedding layer to obtain the initial embeddings per node, (2) GNN to compute a contextual embedding per node, (3) aggregation of individual node embeddings to obtain a graph-level embedding and (4) softmax activation layer to assign a probability to each defined edit operation. (a) the graph-level embedding is used as input by the other two neural modules (i.e., f^j_{conn} and f_{end}) and (b) the selected edit operation is used as input by the neural module f^j_{conn}.

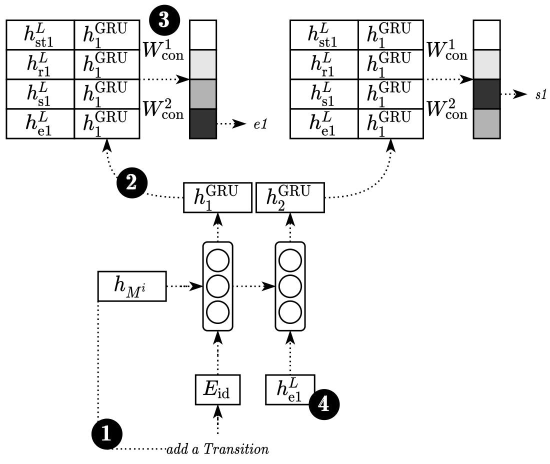


Fig. 9: Architecture of f^j_{conn}.

3.1.3 Architecture of the termination module

Each time that we update the partial model, we need to check whether this is the last edit operation that we need to apply. For this task, the termination module (f_{end}) consists in applying two linear layers with a sigmoid activation function to the input graph embedding (label a in Fig. 8), in order to output the probability of finishing the generation process.

$$f_{\text{end}}(M^i) = \sigma(W^2_{\text{end}} \cdot \text{ReLU}(W^1_{\text{end}} \cdot h_{M^i})).$$

3.2 Inference phase

The inference algorithm is described in Algorithm 1. It takes an initial model and a size threshold and it outputs a complete model. In each iteration, it applies an edit operation to iteratively complete the model. One step of this Algorithm is illustrated in Fig. 10. Given a partial model M^i, to obtain the next model M^{i+1}, we apply these two steps:

1) Sample an edit operation from $f_{\text{edit}}(M^i)$. In this example, the sampled edit operation is to *add a Transition* between the *Entry e1* and the *State s1*. This sampling process is composed by:
   a) Sample from $f_{\text{id}}(M^i)$ the identifier of the next edit operation. The softmax layer assigns a probability to each identifier, so we pick one randomly according to the probabilities.
   b) For each connector in the edit operation, we sample a node of M^i using $f^j_{\text{conn}}(M^i)$.
2) Apply the stopping criteria. We sample from $f_{\text{end}}(M^i)$ whether the process should end or not (i.e., it decides whether the model is already similar enough to the dataset). Alternatively, it is possible to stop when the given size threshold is reached.

Algorithm 1 Inference phase

```
   Input Initial model $M^1$, threshold $\delta$
1: $M \leftarrow M^1$
2: end ← false
3: while $|M| < \delta$ and not end do
4:     $e \leftarrow$ sample from $f_{\text{edit}}(M)$        ▷ $f_{\text{edit}} := f_{\text{id}} || f^j_{\text{conn}}$
5:     end ← sample from $f_{\text{end}}(M)$
6:     $M \leftarrow e(M)$        ▷ Application of the edit operation
7: end while
```

3.3 Training phase

The neural model is trained in the task of predicting the next edit operation given a partial model. Thus, to obtain the training data points (i.e., the input of the neural network and the expected output) we traverse the dataset of real models $\{M_1, \dots, M_n\}$ transforming each M_i in a sequence of edit operations. In particular, we obtain sequences of tuples that are composed by 1) a partial model (i.e., the input of the neural model) 2) the edit operation that will be applied (this include the edit operation identifier and the nodes of the partial graph that will be connected to the edit graph) and 3) a boolean indicating if the generation process must end. We flat the set of sequences of tuples to a set of tuples to get the training data points. Finally, we train the neural model using the cross-entropy loss in each one of the modules f_{id}, f^j_{conn} and f_{end}.

A key part of the training procedure is the decomposition of an input model into a sequence of edit operations. It is worth noting that, for a given model, there is an exponential number of sequences that can be generated, which makes the process intractable in practice. Therefore,

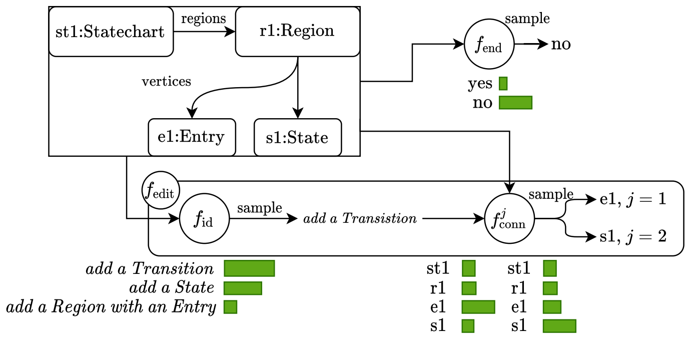


Fig. 10: Application of $f_{\text{edit}} := f_{\text{id}} || f^j_{\text{conn}}$ and f_{end} over the model of Fig. 6.

our approach is to sample k possible sequences for each model in the training set (i.e., *Monte Carlo* approximation).

To sample a concrete sequence, our procedure takes as input a model to be decomposed and the list of edit operations associated to the meta-model. Fig. 11 shows how the graph in Fig. 2 is decomposed as a sequence of tuples consisting of the partial model, the identifier of the applied edit operation and whether or not this is the last edit operation. The algorithm works backwards. Starting with the input model, we iterate until we reach an empty model by removing elements in each iteration. To remove elements, we try to undo an edit operation. Undoing an edit operation requires finding a subgraph in the current model which is isomorphic to the edit graph. Since there could be several possible isomorphic graphs, we pick one randomly. In the example, we can find a subgraph for the *add a Transition* operation in the input model M^7, consisting of $t2$ and connecting the nodes $s1$ and $s2$. We remove $t2$ and its references to obtain the next partial model M^6. Now, we generate a tuple with the identifier of the operation, the new partial model, the connecting nodes ($s1$, $s2$) and whether this is end of the generation process (`true` in this case because this is the first step).

The selection of which edit operation to use is done in priority order (with the aim of reducing the combinatorial space), signifying that we pick the first edit operation in the list and try to apply it. If it is not possible to find a matching subgraph as explained before, we try the next operation until one succeeds. In the example, let us assume that all the *add a Transition* operations has been undone, and the next matching operation is *add a State* (from M^4 to M^3 in Fig. 11). We continue the process and the last operation to undo is *add a Region with an Entry* to reach an empty model and end the process (from M^2 to M^1 in Fig. 11).

The training procedure that uses the algorithm explained before (`sequenceGeneration`) is shown in Algorithm 2. The first group of for loops generates the training data points (lines 1-6) and the second group trains the model using stochastic gradient descent (lines 7-12).

4 Evaluation

In this section we report the results of our evaluation. We have evaluated how *consistent*, *diverse*, *realistic*, and *scalable* is our approach in comparison with other state-of-the-art generators. We have also evaluated whether the generated models produced by our generator are *novel* (unseen in the training set) and *unique* (it does not generate the same models). The experiments presented in this section are available in `https://figshare.com/s/f8b821fb6e72d8986314`. The source code of the standalone version of the M2 generator is available in `https://github.com/Antolin1/M2` to be used as a Python library.

4.1 Datasets

We have considered four datasets as samples of real models. These models belong to three domains (meta-modeling, databases and statecharts). Three of the datasets have been extracted from the MAR search engine [29] and reused from [22]. They are *Ecore-GitHub* (281 models) which is a dataset of Ecore models [22], *Yakindu-GitHub* (129 models) a dataset of Yakindu models [28], and *Rds-GenMyModel* (402 models) a dataset of relational models extracted from GenMyModel[1]. We have also used *Yakindu-Exercise* (274 models), which is a dataset used in [12], consisting of Yakindu models [28] made by students as solutions for similar (but not identical) statechart modeling homework assignments. For all datasets we removed inconsistent models and isomorphic models. The boxplot in Fig. 12 shows the distribution of model sizes. The meta-models and the constraints considered in each domain, as well as the edit operations that we have designed for each meta-model are described in detail in Appendix A.

4.2 Comparing real models against synthetic models

To perform the evaluation, we need a way to assess how the models generated by a generator are with respect to real models. Therefore, we need to create a dataset of synthetic models and compare it with a dataset of real ones. To this end, we have to adjust the parameters (input conditions) of each generator to make it generate models as similar as possible to the real models.

The input conditions of a generator, $\{c_i\}_{i=1}^n = \{f_i\}_{i=1}^p \cup \{v_i\}_{i=1}^q$, can be *fixed* ($f_i$) or *variable* ($v_i$) in the generation process. For instance, in most generators the input meta-model and the constraints are fix conditions, whereas the

1. https://www.genmymodel.com/

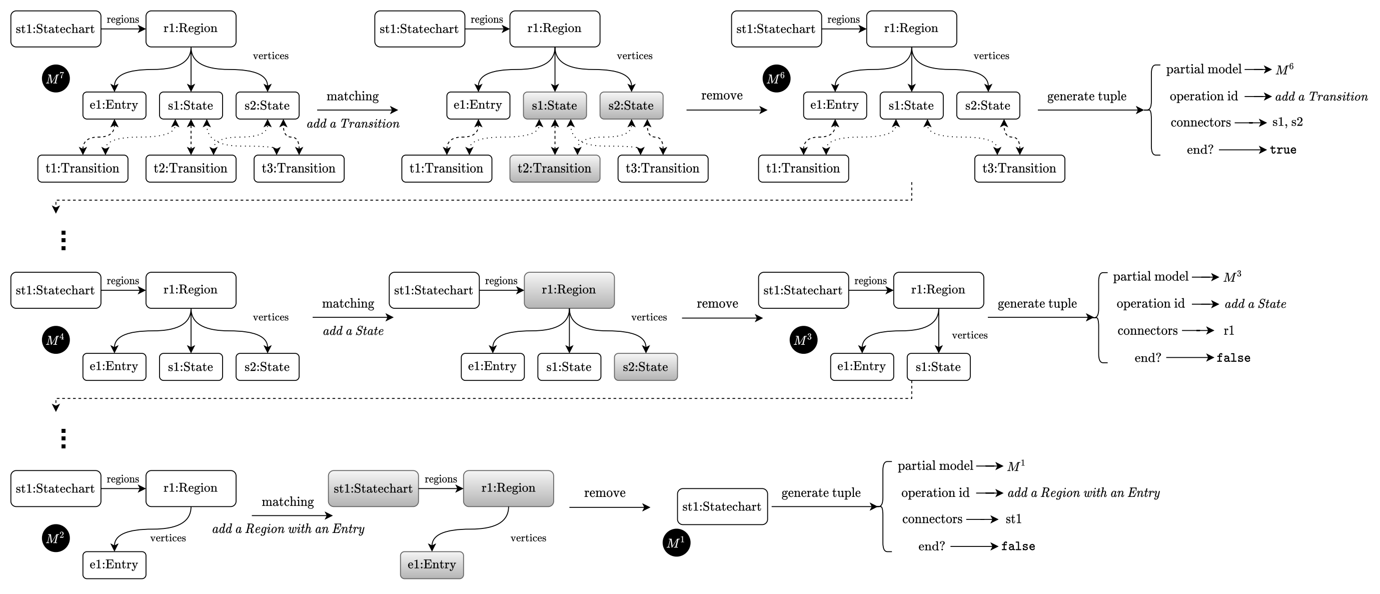


Fig. 11: Decomposition process for the model in Fig. 2.

Algorithm 2 Training procedure

```
    Input Model from the dataset $M$
    Input List of edit graphs/operations $E$
 1: $T \leftarrow \emptyset$
 2: for $i = 1, \dots, k$ do
 3:     for $M \in$ train set do
 4:         $T \leftarrow T \cup$ sequenceGeneration$(M, E)$
 5:     end for
 6: end for
 7: for $e = 1, \dots,$ max epochs do
 8:     $T_{\text{batchs}} \leftarrow$ batchify $T$
 9:     for batch $\in T_{\text{batchs}}$ do
10:         Update $\theta$ w.r.t. $\sum_{(M', e', \mathbb{V}', f') \in T_{\text{batchs}}} \nabla$Cross-entropyLoss$(M', e', \mathbb{V}', f')$
11:     end for
12: end for
```

size of the output model is a variable condition. Therefore, we can define a new distribution (associated to a given generator) which considers fixed and variable conditions separately as:

$$P_{\text{gen}}(M) = \sum_{(v_1, \dots v_q) \in \mathcal{V}} P(v_1, \dots v_q) P(M | v_1, \dots v_q, f_1, \dots, f_p)$$

The sampling process is shown in Fig. 13. To obtain a model, firstly we have to sample $(v_1, \dots, v_q)$ from $P(v_1, \dots v_q)$ (label ❶), introduce the sampled $v_1, \dots, v_q$ and the fixed $f_1, \dots, f_p$ as input to the generator (label ❷) and then use the generator to sample a model M using all the input conditions (label ❸). The reason behind this detour is that, if we want to test the quality of the generator through a comparison using a set of real models, we must estimate $P(v_1, \dots, v_q)$ by using the real dataset (label ⓿) and sample concrete values of $v_1, \dots, v_q$ from it. The expectation is that $v_1, \dots, v_q$ are the best parameters for the generator to produce synthetic models as close as possible to the real ones. Then, we can study the differences between both the real dataset and the synthetic dataset and evaluate the ability of generator to obtain realistic models. This process can be seen as a way of fitting the generator with respect to a given dataset of real models.

We will use this technique to evaluate our proposal with respect to several generators, whose features are described next.

4.3 Baselines

We aim at comparing our model generator against three state-of-the-art generators which are representative of different generation approaches. Taking into account the formulation introduced in the previous section to fit the generator parameters to each dataset, we have considered these three generators as baselines to compare with:

- VIATRA graph generator [1]: It is a search based generator that receives a meta-model $\mathcal{M}$, the number of objects (o) that the output model will have and a set of well-formedness constraints Ψ. It can be seen as this distribution over models:

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

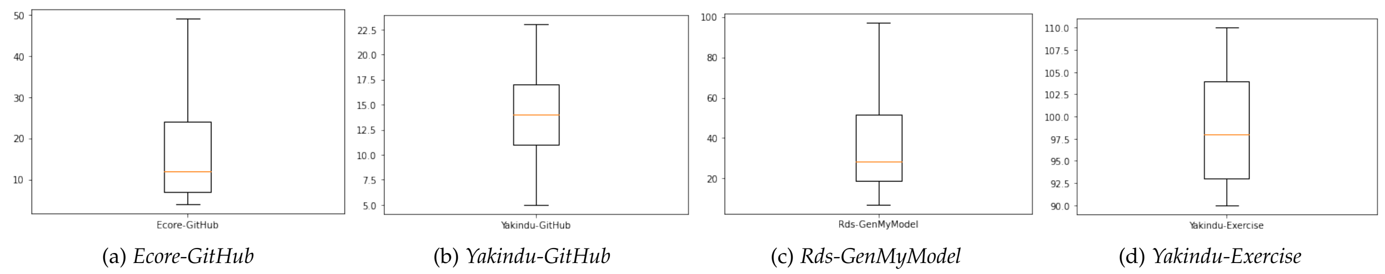


Fig. 12: Boxplots (without outliers) of the number of objects in each dataset.

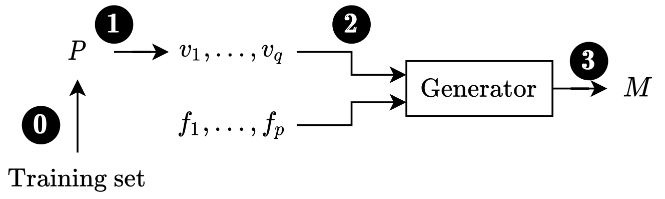


Fig. 13: Estimation of P and sample procedure associated to P_{gen}.

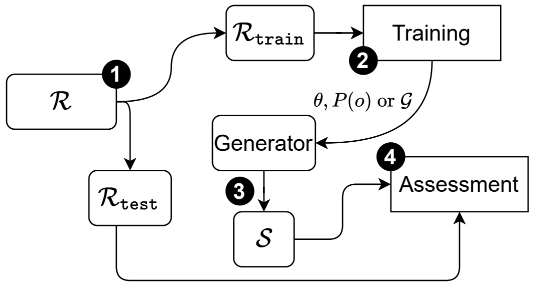


Fig. 14: Experimental procedure. $\mathcal{R}$ is the dataset of real models. $\mathcal{R}_{\texttt{train}}$ and $\mathcal{R}_{\texttt{test}}$ are the subsets of $\mathcal{R}$ used to fit the generators and perform the assessment respectively. $\mathcal{S}$ represents the set of generated models.

The sampling process in this case requires establishing the meta-model and the domain constraints as fix input conditions and the size of the model as variable condition (named o). Thus, given a dataset of real models, we can estimate $P(o)$ in order to know which are the sizes of the real models. Then, to construct a synthetic dataset of n models we first sample n sizes from $P(o)$ and invoke the VIATRA generator n times. In this way, we have constructed a dataset of synthetic models as similar as possible to the real ones, using the degrees of freedom provided by the generator.

- EMF random instantiator (RANDOM): This generator receives a meta-model $\mathcal{M}$, the number of objects (o) that the output model will have and the average number of references per object (d). It does not support domain constraints so the generated models could not be consistent. We fix $\mathcal{M}$ to the corresponding meta-model and d to its default value. We consider o as variable. Therefore, this generator can be seen as this distribution over models:

$$P_{\text{gen}}(M) = \sum_{o \in O} P(o) P_{\text{RANDOM}}(M|\mathcal{M}, o, d)$$

- RandomEMF [10] (rEMF): It is a grammar-based random generator. It receives a meta-model $\mathcal{M}$ and a grammar $\mathcal{G}$ that guides the generation process.

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

4.4 Experimental procedure

To make the generators comparable and to be able evaluate them with respect to a dataset of real models $\mathcal{R} = \{M_1, \ldots, M_n\}$, we adjust the parameters of each generator as explained above and follow the procedure depicted in Fig. 14. This method is similar to the one proposed in [22] and is summarized as follows:

1) $\mathcal{R}$ is split into $\mathcal{R}_{\texttt{train}}$ and $\mathcal{R}_{\texttt{test}}$ (label ❶). In our experiments we use 60% for training and 40% for testing.
2) $\mathcal{R}_{\texttt{train}}$ is used to train the generators (label ❷), that is, to fit their parameters to $\mathcal{R}_{\texttt{train}}$:
   - To train RANDOM and VIATRA we follow the ideas of [22], that is, we approximate $P(o)$ by using Kernel Density Estimation (KDE) with the gaussian kernel. The bandwidth (which is a hyperparameter of the KDE) is estimated with the Improved Sheather Jones method [35].
   - To train rEMF, that is, to estimate $\mathcal{G}$ we use the same approach as [22]. It consists in building the rules manually and using maximum likelihood estimation to estimate the parameters of the rules.
   - To train our model, that is, to estimate θ (all the parameters of the neural network) we use the Algorithm 2.
3) Once the generator is trained, we use it to generate $|\mathcal{S}| = 500$ models (label ❸).
4) The set $\mathcal{S}$ together with $\mathcal{R}_{\texttt{test}}$ will be used to perform the assessments (label ❹).

4.5 Assessing Consistency

To evaluate consistency, for each pair of dataset and generator we study the proportion of inconsistent models in the set $\mathcal{S}$ of generated models. The results are reported in Table 1.

As it is expected, RANDOM obtains the worst results. Furthermore, this generator does not produce any consistent model in the case of Yakindu because of its inability to take into account the associated constraints. VIATRA is a

	Ecore-GitHub	*Yakindu-GitHub*	*Rds-GenMyModel*	*Yakindu-Exercise*
RANDOM	4.2%	0%	6.4%	0%
rEMF	100%	48.2%	93.4%	60.4%
VIATRA	100%	100%	100%	100%
M2	99.8%	90.8%	99.4%	87.0%

TABLE 1: Proportion of consistent models for each generator and for each dataset.

	Ecore-GitHub	*Yakindu-GitHub*	*Rds-GenMyModel*	*Yakindu-Exercise*
RANDOM	23.8	∞	14.6	∞
rEMF	1	2.1	1.1	1.7
VIATRA	1	1	1	1
M2	$\sim$1	1.1	$\sim$1	1.1

TABLE 2: Expected number of runs until a consistent model is reached.

consistent model generator so all the generated models are consistent. rEMF obtains similar results in comparison with our generator in *Ecore-GitHub* and *Rds-GenMyModel* (but at the cost of manually tweaking the grammars to make sure that few constraints are violated). However, in the case of Yakindu, ours works significantly better.

It is possible to overcome the limitation of a generator not achieving full consistency. To turn a partially consistent generator into a fully consistent one, it is enough to run it multiple times until the required number of consistent models are obtained, discarding the inconsistent ones. Thus, in Table 2 we report the expected number of runs to obtain a consistent model for each generator and for use-case. This table was computed using Table 1 and the formula $1 + \frac{1-p}{p}$ where p is the probability of obtaining a consistent model from the generator (i.e., cells of Table 1). The random variable associated to a generator and defined as the number inconsistent models until a consistent one is obtained follows a geometric distribution. This is because, each run can be seen as an independent Bernoulli trial.

Assessment: Our generator is not consistent, that is, there is no guarantee that the generated models respect the domain constraints. The only truly consistent generator is VIATRA. However, the majority of the models produced by M2 are consistent and we can conclude that M2 is *almost consistent*. However, it is possible to turn M2 into a fully consistent generator by performing a few more extra invocations.

4.6 Assessing Novelty and Uniqueness

Before comparing with the other generators in terms of diversity and realism, we have studied whether the models generated by M2 are novel (that is, not seen in training data) and unique (the generator does not always generate the same models). To this end, we remove the inconsistent models from the set of generated models $\mathcal{S}$ and calculate the percentage of unique and novel models:

$$\text{PUNM} = \frac{|\textit{uniques}(\text{CON} \cap \text{NOV})|}{|\text{CON}|} \times 100\%$$

where CON denotes the subset of consistent models of $\mathcal{S}$, the function *uniques* receives as input a set of models and removes the duplicates (isomorphic) and NOV denotes the models in $\mathcal{S}$ that are unseen in the train set.

Table 3 shows the percentage of unique and novel models over the set of consistent generated models for each one of the four datasets achieved by our generator. The worst case corresponds to the dataset *Yakindu-GitHub*. This is caused by the fact that the real dataset is composed by relatively small models, so it is easy to overfit the training data (lower |NOV|) and it is more likely to repeat models in the generation process (lower $|\textit{uniques}(\text{CON} \cap \text{NOV})|$). On the other hand, the situation in *Yakindu-Exercise* is the opposite. This dataset is more difficult to overfit since it is composed by large models (higher $|\textit{uniques}(\text{CON} \cap \text{NOV})|$).

	PUNM
Ecore-GitHub	85.8%
Yakindu-GitHub	77.3%
Rds-GenMyModel	95.2%
Yakindu-Exercise	100%

TABLE 3: Percentage of unique and novel models over the set of consistent generated models for each one of the four datasets.

Assessment: The percentage of novel and unique models over the set of consistent models generated by our generator is typically high.

4.7 Assessing Diversity

To study the diversity of the generated models we use the *internal diversity* metric proposed in [18]. It is defined by $0 \leq d_i^{\text{int}} = \frac{|S_i(M)|}{|M|} \leq 1$, where $|M|$ is the number of objects of the model and $|S_i(M)|$ is the number of neighborhood shapes of range i. For each real and synthetic model, we compute its internal diversity fixing $i = 5$ (as it is done in [12]). We remove the inconsistent models from the set of generated models. The rationale is to avoid the bias produced by shapes that are not allowed (since they do not respect the domain constraints).

The results are shown in Table 4. We compare the internal diversity of the models generated by M2 with the dataset of real ones by using the $t-$test. We have found differences in *Ecore-GitHub* and *Yakindu-Exercise*. In those datasets, synthetic models have more internal diversty. In *Yakindu-GitHub* and *Rds-GenMyModel*, we have not found differences.

Assessment: Baselines are more diverse than the real models and outperform our generator in this property. However, M2 is at least, as diverse as the real models. Thus, if the dataset is diverse, our generator will be diverse.

	Ecore-GitHub	*Yakindu-GitHub*	*Rds-GenMyModel*	*Yakindu-Exercise*
RANDOM	0.265 ± 0.063	-	0.571 ± 0.140	-
rEMF	0.793 ± 0.141	0.900 ± 0.106	0.614 ± 0.177	0.740 ± 0.145
VIATRA	0.748 ± 0.176	0.889 ± 0.102	0.582 ± 0.177	0.857 ± 0.056
M2	$0.724 \pm 0.214^{\dagger}$	0.836 ± 0.159	0.385 ± 0.231	$0.578 \pm 0.151^{\dagger}$
Real models	0.644 ± 0.262	0.866 ± 0.169	0.428 ± 0.231	0.507 ± 0.079

TABLE 4: Averages $\pm$ standard deviations of the internal diversity for each generator and for each dataset. In the M2 row, the symbol † indicates that there are statistically significant differences (using $t-$test) when comparing with the real models ($p-$value< 0.01).

4.8 Assessing Realism

To assess the sample quality of the generators in terms of realism, we use the Maximum Mean Discrepacy [26] (MMD) over graph metrics [27] rather than simple comparisons of statistics [5], [12]. This method is widely used to assess generative models of graphs [24], [27], [36] and compares all moments of their empirical distributions.

One use-case of the MMD is to solve the two-sample test [26], i.e., given two set of samples, determine whether these sets come from the same distribution. In this case, MMD can be seen as a type of metric that compares the two distributions that generate these two sets of samples and, the closer to zero, the more similar the distributions are. This metric is flexible (it can deal with any type of samples not only numeric) because it relies on a kernel.

In our setting, given the set $\mathcal{R}_{\texttt{test}} = \{M_1, \ldots, M_n\}$ and the set $\mathcal{S} = \{M'_1, \ldots, M'_m\}$, we extract the graph distribution of each model obtaining $\mathbb{D}_{\texttt{test}} = \{g_1, \ldots, g_n\}$ and $\mathbb{S} = \{g'_1, \ldots, g'_m\}$. Where each g_i, g'_j are univariate distributions over $\mathbb{R}$ (such as node distribution, clustering coefficient, etc). Let us suppose that $\mathbb{D}_{\texttt{test}}$ comes from the distribution $\boldsymbol{g}$ and $\mathbb{S}$ comes from $\boldsymbol{g'}$. We want to know how close are $\boldsymbol{g}$ and $\boldsymbol{g'}$ so we compute the MMD^2:

$$MMD^2(\boldsymbol{g}||\boldsymbol{g'}) = \mathbb{E}_{g_1,g_2 \sim \boldsymbol{g}}\left[k(g_1,g_2)\right] + \mathbb{E}_{g'_1,g'_2 \sim \boldsymbol{g'}}\left[k(g'_1,g'_2)\right] - \\ -2\mathbb{E}_{g,g' \sim \boldsymbol{g},\boldsymbol{g'}}\left[k(g,g')\right].$$

Here, k is a kernel function. We use the kernel proposed in [27] which uses the first Wasserstein distance to compare two distributions.

Regarding the graph distributions, we consider Multiplex Participation Coefficient (MPC), Normalized Node Activity (NNA) and Degree Distribution (DD) used in previous works to assess the realistic property of model generators [5], [12], [21]. The definitions of MPC and NNA are shown in Appendix B for completeness.

As it is done when assessing diversity, we remove inconsistent models from the set of generated models and we report the MMD^2 for each generator, dataset and graph metric. The results are reported in Table 5. The generator prosed in this work outperforms all the baselines in all domains since it obtains, for each metric, the lowest MMD^2.

Assessment: Our generator, M2, outperforms all the baselines in the realistic property. The results indicate that the models generated by our generator tend to be structurally similar to the ones on the corresponding datasets.

4.9 Assessing Scalability

We are interested on evaluating the scalability of our generator, taking into account the other baselines. To this end, for each generator and dataset we produce 500 models following a distribution of model sizes. Each distribution is estimated to reflect the sizes of the corresponding dataset. For each invocation of a generator, we measure the time it takes to produce the model and the size of the produced model. Then, to present the results, we group the generated models by ranges of sizes and take the average time. The ranges were calculated by splitting the interval $[\text{min}_{\texttt{test}}, \text{max}_{\texttt{test}}]$ (where $\text{min}_{\texttt{test}}$ is the minimum size of the models in the test set and $\text{max}_{\texttt{test}}$ the maximum one) in three intervals of approximately the same size.

The results are shown in Table 6. It can be observed that rEMF and RANDOM have a very good performance, with average times around a few milliseconds to generate a single model[2]. However, M2 and VIATRA present large execution times since their design is intended to generate richer models, thus they are compared against each other. M2 performs better than VIATRA in the Yakindu and Ecore domain. In the case of *Rds-GenMyModel*, VIATRA is faster in the ranges $[7, 80]$ and $[81, 153]$. However, in the range $[154, 227]$, M2 is faster. We believe that this is caused by the fact that the generation time of VIATRA grows exponentially, while the M2 grows polynomially. This can be observed by looking at the *log-log* scatter plot of the execution time with respect to the number of elements. For example, in the case of *Ecore-GitHub* (Fig. 15) and *Rds-GenMyModel* (Fig. 16), we can see that the M2 generation time follows a straight line whereas the VIATRA generation time follows an exponential line. This can be caused by the fact that VIATRA is a search-based generator and the search space grows exponentially. On the other hand, the functions f_{edit} and f_{end} in Algorithm 1 (generation algorithm of M2) are neural networks so the forward procedure is polynomial.

One shortcoming of M2 is that it has to be trained to be able to generate models. However, this process is only executed once. Table 7 shows the training time for each dataset which includes the generation of the training datapoints (set T in Algorithm 2). The training is performed on a single GPU NVIDIA GeForce RTX 2060 and the hyperparameters used are shown in Appendix C.

2. We found that RANDOM has problems when generating small models due to its generation algorithm, which is designed to generate large models. Because of that, RANDOM did not produce models in between the intervals $[6, 11]$ and $[12, 16]$ in the case of *Yakindu-GitHub*.

	Ecore-GitHub			*Yakindu-GitHub*			*Rds-GenMyModel*			*Yakindu-Exercise*		
	DD	NNA	MPC	DD	NNA	MPC	DD	NNA	MPC	DD	NNA	MPC
RANDOM	1.009	0.639	1.044	-	-	-	0.541	0.130	0.549	-	-	-
rEMF	0.078	0.017	0.018	0.087	0.103	0.171	0.137	0.083	0.166	0.309	0.222	0.356
VIATRA	0.145	0.096	0.139	0.469	0.596	0.789	0.201	0.089	0.442	0.538	0.257	0.362
M2	**0.019**	**0.010**	**0.016**	**0.015**	**0.002**	**0.004**	**0.015**	**0.007**	**0.011**	**0.276**	**0.006**	**0.004**

TABLE 5: MMD^2 for each generator, dataset and graph metric.

	Ecore-GitHub			*Yakindu-GitHub*			*Rds-GenMyModel*			*Yakindu-Exercise*		
	[4,63]	[64,123]	[124,184]	[6,11]	[12,16]	[17,22]	[7,80]	[81,153]	[154,227]	[90,96]	[97,103]	[104,110]
RANDOM	0.009	0.013	0.014	-	-	0.009	0.011	0.011	0.015	0.011	0.012	0.012
rEMF	**<0.001**	**<0.001**	**0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**0.001**	**0.003**	**<0.001**	**<0.001**	**<0.001**
VIATRA	0.657	1.255	15.869	0.221	0.389	0.192	**0.132**	**0.465**	1.965	1.073	0.902	1.030
M2	**0.088**	**0.999**	**1.951**	**0.023**	**0.039**	**0.059**	0.181	0.777	**1.868**	**0.681**	**0.756**	**0.838**

TABLE 6: Average generation time (in seconds) for each generator, dataset and size split. The symbol '-' means that there are no models whose sizes belongs to the interval.

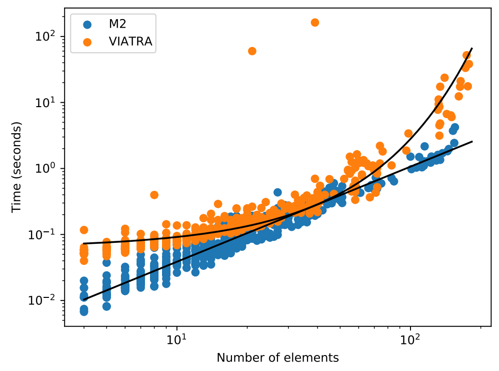


Fig. 15: Generation time vs size for each one of the generators in *Ecore-GitHub*.

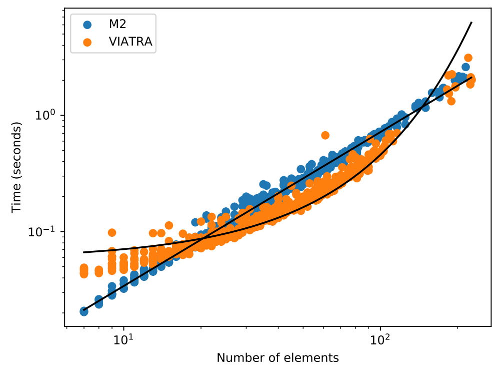


Fig. 16: Generation time vs size for VIATRA and M2 in *Rds-GenMyModel*.

Assessment: Both rEMF and RANDOM are better in terms of scalability than VIATRA and M2. Comparing M2 with VIATRA, the former is better in the inference time because the VIATRA generation time grows exponentially with respect to the size of the produced models. A shortcoming of M2 is the fact that it must be trained before the generation.

	Training time (min)
Ecore-GitHub	~ 10
Yakindu-GitHub	~ 1
Rds-GenMyModel	~ 26
Yakindu-Exercise	~ 14

TABLE 7: Training phase duration.

4.10 Global Assessment

This work was aimed to design a generator focused on satisfying the realistic property and we have shown that M2 outperforms the baselines in this property. However, it also achieves acceptable results in the other properties. In particular, M2 is *almost consistent* with proportions of consistent models greater than 85%. Regarding diversity, by design we cannot ask the generator to be more diverse than the dataset it tries to imitate. But we can claim that the diversity property is reached if the source dataset is diverse. One shortcoming of M2 is that it has to be trained before being able to produce models, but its inference procedure is scalable.

A distinctive feature of M2 is that it focuses on realism at the expense of sacrificing a bit some of the other properties. In the evaluation, we have shown that, with our approach, we can achieve results *almost* similar to state-of-the-art generators but supporting realism, a property which has been neglected until now. In particular, the fact that consistency is not fully achieved is not an important issue because we can repeatedly invoke the generator until the required consistent models are obtained and discard the inconsistent ones.

Regarding the user effort to use each generator, it is worth mentioning that rEMF depends on a grammar $\mathcal{G}$ that is defined by the user and guides the generation process. To perform the experiments, we put a lot of effort when designing the rules in order to respect the domain constraints and provide realistic models. The result of this effort is a difficult grammar composed by some unintuitive twisted rules. Therefore, the measures of consistency and the realism reported in this paper could be optimistic in the case of rEMF. On the other hand, for M2 the user only needs to define edit operations, whose definition is intuitive and fast to establish since it is what a user performs when she is making a model (e.g., *add a Transition*, *add a State*, etc).

It is important to note that the edit operations can be defined automatically by extracting atomic operations from

	Consistency	Diversity	Realism	Scalability
RANDOM	○○○○	○○○○	○○○○	●●●●
rEMF	●●●○	●●●●	●●○○	●●●●
VIATRA	●●●●	●●●●	●○○○	●●●○
M2	●●●○	●●○○	●●●●	●●●○

TABLE 8: Summary of the four properties for each one of the generators.

the meta-model. However, if all edit operations are atomic then the generator may struggle in terms of consistency. Therefore, we recommend using the domain knowledge to design more complex edit operations and put them on the top of the priority list. With the current implementation of M2, the complex edit operations have to be designed manually. Nevertheless, this design is not necessarily difficult. One can use as reference existing graphical editors, APIs and even the associated meta-model constraints to take inspiration to define these edit operations. For instance, in the use-cases presented in this paper, we have used the graphical editors of EMF [37], Yakindu [28], and GenMyModel as references to derive complex edit operations (only 3, 2, and 1 complex edit operations were required in each case). On the other hand, we believe that this type of operations can be derived directly from the meta-model and the associated constraints by using logic analysis (SAT solvers). We plan to tackle this problem in the future.

A limitation of our generator is that the size of the output model is difficult to be controlled by the user in the inference phase. This is because stopping criteria in the Algorithm 1 is given by a threshold δ and the module f_{end} (which is a black box neural network). Thus, the size of the output can be any size in the interval $[|M^1|, \delta + k]$ (where k is the number of new nodes added by the largest edit graph). Two ways to solve this problem are *twisting* the Algorithm 1 with some heuristic or trying to learn a conditional probability $P_\theta(M|o)$ (where o is desired size of the output model). We left the exploration of these two solutions as future work.

Theoretically, M2 learns the distribution of real models. Thus, if the training models satisfy a set of constraints or graph properties, the generated graphs tend to satisfy them. For instance, let us suppose that we have a constraint which establishes that the number of objects has to be prime. In this case, the training samples must also have sizes of five or seven. After training, the generated models will most probably have sizes of five and seven. This is because M2 tries to imitate the distribution of real models which also includes the sizes (five and seven). However, it is unlikely that the generator produces models with size 121, since this type of valid models has not been seen in the training. It is important to remark that the statistical graph properties that M2 could recognize and reproduce is given by the expressiveness of the neural model. In this case, it has been shown that every graded modal logic formula defined over nodes is expressible by a GNN [38], [39].

Table 8 shows a summary of the assessment of the four properties for each generator. The worst generator is RANDOM which stands out in scalability but performs poorly in the other properties. The other generators have weaknesses and strengths, but they are applicable depending on the property of interest. For instance, rEMF is good for diversity and scalability, VIATRA is a perfect option when the user looks for consistency, and diversity, whereas M2 is a good choice when the user is interested in realism but also having a good balance with respect to diversity, consistency and scalability.

5 Related work

In this section we review some works related to our proposal, organized in six categories: SAT-solver based model generators, generators based on constructive formalisms, search-based model generators, traditional random models of graphs, deep generative models of graphs and assessing realism in model generators.

SAT-solver based model generators. The underlying idea of these generators is to map the meta-model and the constraints to logical formulas. Consistent models are obtained by using SAT-solvers over these formulas. Therefore, it is possible to build these type of generators using model finders like Formula Framework [9], Alloy [7] and EMF2CSP [8]. The properties of Alloy-based SAT solvers have been evaluated with respect to VIATRA Generator in a number of works [5], [12], [18], [40], showing that Alloy produces consistent models, but struggles at diversity, realism and scalability.

Generators based on constructive formalisms. The idea of these generators is to use formal grammars or graph grammars to guide the generation process. Examples are the generator proposed in [11] (that uses graph grammars), RandomEMF [10], and [30]. In particular, the work in [30] uses a rule-based approach to generate models by consecutively applying transformation rules. Some of these rules are derived from the meta-model and they are very similar to our atomic edit operations.

Search-based model generators. The generation problem is transformed into a search problem whose target is a model that respect the constraints and the meta-model. An example is VIATRA solver [1] (and its later improvements [12], [41]) which is more scalable than SAT-solver based generators [1], [5] since it uses a scalable graph engine. Another example is SDG [13]. It proposes a search-based custom OCL solver approach to generate synthetic data for statistical testing and it is used later in [14] to synthetize test data in UML.

Traditional random models of graphs. Random models of graphs like Erdös-Rényi [42] or Watss-Strogatz [43] were proposed to model the networks that are in nature. They can also be considered model generators but the graphs that they produce are unlabeled.

Deep generative models of graphs. The complexity of the networks that we can find in nature cannot be fully explained by the traditional random models. Therefore, some generative models of graphs based on deep learning have been proposed. There are basically two types of generative methods [24]: one-shot generating and sequential generating. One-shot learning generates full graph graph in one shoot. For example, GraphVAE [44] that uses variational autoencoders belongs to this group. On the other hand, sequential generating aims to generate nodes and edges in a sequential way. Our generator belongs to this group.

Some examples are: GraphRNN [27], which uses a two-level recurrent neural network to generate a graph; [45] considers a generation process in which each step can be to add a new node or add a new edge. [33] tackles the molecule generation process as a sequence of transitions (adding a new atom or connect an existing one to another). Our approach is very similar to this work but there are two differences, firstly the neural model used is not the same (our neural model is prepared for dealing with labeled edges) and secondly, the edit operations can be seen as a generalization of their defined transitions. Similarly, the work by You et al. [46] approaches the generation process as a Markov Decision Process and tries to optimize molecule properties while ensuring realism.

Assessing realism in model generators. The authors in [21] propose to use multidimensional graph metrics to characterize realistic models. In particular, Varró [5] uses the average KS statistic of these metrics between all the synthetic models and all the real models. As it is pointed out in [22], this metric does not take into account dissimilarities inside the set of real models and inside the set of generated models so it is not consistent. In this paper, we propose to use the MMD [26], [27] over the graph statistics proposed in [21] to measure how realistic a generator is. It is similar to the method of [5] but it is consistent. Finally, [22] proposes to use GNNs together with C2ST [47] to assess the realistic property. We decide not to use this technique here because of the small size of the test set.

6 Conclusion and future work

In this paper we have proposed a new model generator, M2, fully focused on satisfying the structurally realistic property. It relies on a deep autoregressive model which constructs a model by sampling an edit operation in each step of the generation process. It is able to imitate the structure of the training models but generating novel ones. Furthermore, in this work we have evaluated M2 and we have compared it with three state-of-the-art generators. We concluded that our generator outperforms the other generators in the realistic property, the majority of the generated models are consistent, the diversity of the generated models is similar to the real ones and the generator process is scalable once the generator is trained.

As future work, we plan to derive complex edit operations automatically from the meta-model and tackle the limitation of the inference phase with respect to the size of the output models We also want to make the generator fully realistic by considering attribute values. Finally, we want to explore the integration of M2 into another generator, like VIATRA generator, in order to enhance properties like consistency and diversity.

Acknowledgments

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.

References

[1] O. Semeráth, A. S. Nagy, and D. Varró, "A graph solver for the automated generation of consistent domain-specific models," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 969–980.

[2] G. Bagan, A. Bonifati, R. Ciucanu, G. H. Fletcher, A. Lemay, and N. Advokaat, "gmark: Schema-driven generation of graphs and queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 4, pp. 856–869, 2016.

[3] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE, 2001, pp. 22–31.

[4] Z. Micskei, Z. Szatmári, J. Oláh, and I. Majzik, "A concept for testing robustness and safety of the context-aware behaviour of autonomous systems," in *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*. Springer, 2012, pp. 504–513.

[5] D. Varró, O. Semeráth, G. Szárnyas, and 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.

[6] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 632–647.

[7] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[8] C. A. González, F. Büttner, R. Clarisó, and J. Cabot, "Emftocsp: A tool for the lightweight verification of emf models," in *2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*. IEEE, 2012, pp. 44–50.

[9] E. K. Jackson, T. Levendovszky, and D. Balasubramanian, "Reasoning about metamodeling with formal specifications and automatic proofs," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 653–667.

[10] M. Scheidgen, "Generation of large random models for benchmarking." *BigMDE@ STAF*, vol. 1406, pp. 1–10, 2015.

[11] K. Ehrig, J. M. Küster, and G. Taentzer, "Generating instance models from meta models," *Software & Systems Modeling*, vol. 8, no. 4, pp. 479–500, 2009.

[12] O. Semeráth, A. A. Babikian, B. Chen, C. Li, K. Marussy, G. Szárnyas, and D. Varró, "Automated generation of consistent, diverse and structurally realistic graph models," *Software and Systems Modeling*, pp. 1–22, 2021.

[13] G. Soltana, M. Sabetzadeh, and L. C. Briand, "Synthetic data generation for statistical testing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 872–882.

[14] ——, "Practical constraint solving for generating system test data," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 2, pp. 1–48, 2020.

[15] "AtlanMod Team (Inria, Mines-Nantes, Lina), EMF random instantiator." [Online]. Available: https://github.com/atlanmod/mondo-atlzoo-benchmark/tree/master/fr.inria.atlanmod.instantiator

[16] O. A. Specification, "Object constraint language," 2006.

[17] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró, "A graph query language for emf models," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2011, pp. 167–182.

[18] O. Semeráth, R. Farkas, G. Bergmann, and D. Varró, "Diversity of graph models and graph generators in mutation testing," *International Journal on Software Tools for Technology Transfer*, vol. 22, no. 1, pp. 57–78, 2020.

[19] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing vision-based control systems using learnable evolutionary algorithms," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1016–1026.

[20] M. Z. Iqbal, A. Arcuri, and L. Briand, "Environment modeling and simulation for automated testing of soft real-time embedded software," *Software & Systems Modeling*, vol. 14, no. 1, pp. 483–524, 2015.

[21] G. Szárnyas, Z. Kővári, Á. Salánki, and D. Varró, "Towards the characterization of realistic models: evaluation of multidisciplinary graph metrics," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, 2016, pp. 87–94.

[22] J. A. H. López and J. S. Cuadrado, "Towards the characterization of realistic model generators using graph neural networks," in *Proceedings of the 24rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2021.
[23] M. Dalal, A. C. Li, and R. Taori, "Autoregressive models: What are they good for?" *arXiv preprint arXiv:1910.07737*, 2019.
[24] X. Guo and L. Zhao, "A systematic survey on deep generative models for graph generation," *arXiv preprint arXiv:2007.06686*, 2020.
[25] T. Kehrer, G. Taentzer, M. Rindt, and U. Kelter, "Automatically deriving the specification of model editing operations from meta-models," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2016, pp. 173–188.
[26] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola, "A kernel two-sample test," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 723–773, 2012.
[27] J. You, R. Ying, X. Ren, W. Hamilton, and J. Leskovec, "Graphrnn: Generating realistic graphs with deep auto-regressive models," in *International Conference on Machine Learning*. PMLR, 2018, pp. 5708–5717.
[28] "Yakindu statechart tools." [Online]. Available: https://www.itemis.com/en/yakindu/state-machine/
[29] J. A. H. López and J. S. Cuadrado, "Mar: A structure-based search engine for models," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2020, pp. 57–67.
[30] A. Rule-Based, "Generating large emf models efficiently," *Fundamental Approaches to Software Engineering LNCS 12076*, p. 224, 2020.
[31] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.
[32] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.
[33] Y. Li, L. Zhang, and Z. Liu, "Multi-objective de novo drug design with conditional graph generative model," *Journal of cheminformatics*, vol. 10, no. 1, pp. 1–24, 2018.
[34] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European semantic web conference*. Springer, 2018, pp. 593–607.
[35] S. J. Sheather and M. C. Jones, "A reliable data-based bandwidth selection method for kernel density estimation," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 53, no. 3, pp. 683–690, 1991.
[36] R. Liao, Y. Li, Y. Song, S. Wang, C. Nash, W. L. Hamilton, D. Duvenaud, R. Urtasun, and R. S. Zemel, "Efficient graph generation with graph recurrent attention networks," *arXiv preprint arXiv:1910.00760*, 2019.
[37] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.
[38] M. Grohe, "The logic of graph neural networks," in *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2021, pp. 1–17.
[39] P. Barceló, E. V. Kostylev, M. Monet, J. Pérez, J. Reutter, and J.-P. Silva, "The logical expressiveness of graph neural networks," in *8th International Conference on Learning Representations (ICLR 2020)*, 2020.
[40] O. Semeráth, A. A. Babikian, S. Pilarski, and D. Varró, "Viatra solver: a framework for the automated generation of consistent domain-specific models," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 43–46.
[41] K. Marussy, O. Semeráth, and D. Varró, "Automated generation of consistent graph models with multiplicity reasoning," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–1, 2020.
[42] P. Erdős and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.
[43] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.
[44] M. Simonovsky and N. Komodakis, "Graphvae: Towards generation of small graphs using variational autoencoders," in *International Conference on Artificial Neural Networks*. Springer, 2018, pp. 412–422.
[45] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia, "Learning deep generative models of graphs," *arXiv preprint arXiv:1803.03324*, 2018.
[46] J. You, B. Liu, R. Ying, V. Pande, and J. Leskovec, "Graph convolutional policy network for goal-directed molecular graph generation," *arXiv preprint arXiv:1806.02473*, 2018.
[47] D. Lopez-Paz and M. Oquab, "Revisiting classifier two-sample tests," *arXiv preprint arXiv:1610.06545*, 2016.