[1]

A DSL for Encoding Models for Graph-Learning Processes

Zahra Rajaei[1], Shekoufeh Kolahdouz-Rahimi[1], Massimo Tisi[2] and Frédéric Jouault[4]

[1]*MDSE Research Group, Department of Software Engineering, University of Isfahan, Iran*
[3]*IMT Atlantique, LS2N (UMR CNRS 6004), Nantes, France*
[4]*ERIS, ESEO-TECH, Angers, France*

Abstract

Specific deep-learning tools for graph-structured data, i.e. graph-learning, are successfully used in several domains. Their use in Model-Driven Engineering (MDE) requires MDE practitioners to have a good understanding of technical aspects of the graph-learning process. For instance, automatic translators need to be developed, in order to encode models in the most effective input format for deep-learning neural networks.

With this work, we aim at assisting MDE practitioners in applying deep learning on their models. For this purpose, we introduce a Domain-Specific Language (DSL) for configuring the encoding of models into suitable input for graph-learning tools. This DSL is interpreted to automatically translate MDE datasets, enabling their use in machine-learning pipelines. To evaluate this research, we consider the AIDS dataset as instances of a corresponding metamodel. We use our DSL to automatically encode models of this dataset into the format expected by a graph-learning tool. The experimental evaluation demonstrates that we are able to obtain the same encoding used in related work.

Keywords

Model-Driven Engineering, Model Encoding, Graph Learning, Machine Learning

CEUR-WS.org/Vol-2999/oclpaper5.pdf

1. Introduction

Model-Driven Engineering (MDE) reduces the complexity of current large-scale software by leveraging models to capture domain-specific knowledge [1]. Models and metamodels, i.e., abstract representations of typically domain-specific knowledge, are the primary artifacts. The larger the problem, the more diverse, and the more complex are these artifacts [2]. These models needs to be analyzed for some tasks like classification, clustering, repairing, etc. Artificial intelligence (AI) and machine learning (ML) techniques, especially deep learning (DL), can help software engineers in the field of MDE to manage and analyze various models more easily, quickly, and possibly with fewer errors. ML can enable prediction, or discovery of new patterns, using model-structured data as feature information.

[1]Copyright (c) 2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

OCL'21: 20th International Workshop on OCL and Textual Modeling, June 25 2021, Bergen, Norway

z.rajaei@eng.ui.ac.ir (Z. Rajaei); sh.rahimi@eng.ui.ac.ir (S. Kolahdouz-Rahimi); massimo.tisi@imt-atlantique.fr (M. Tisi); frederic.jouault@eseo.fr (F. Jouault)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

Models are a kind of graph and several flexible frameworks for deep graph learning have been proposed in the literature. PyTorch Geometric [3] and Deep Graph Learning (DGL) [4] are the most popular Graph Neural Network (GNN) frameworks. Both are based on the PyTorch deep learning library. GNNs are a deep learning method to infer meaningful patterns on graph-described data [5]. GNNs have been successfully applied to a broad range of processes to solve a variety of challenges due to the various possibilities provided by graph machine learning and the vast number of applicable tasks on graphs. For example, graph classification, which is the identification of graph class labels according to structural features, is an important task in various domains such as social network analysis [6], and chemoinformatics [7]. Molecules, for instance, can be represented as graphs in chemoinformatics, with nodes representing atoms and edges indicating the existence of chemical bonds between pairs of atoms. Predicting the class label of each graph can then be the desired task. For this purpose, the graph features are analyzed and the relationship between the features and the target class is identified.

The primary challenge in ML for models is to figure out how to represent or encode the model structure so that ML can be effectively applied. In this paper, we propose a Domain-Specific Language (DSL) for transforming models into valid input graphs for GNNs. We use the word ”encoding” throughout the paper as as an idiom for ”model to graph transformation”. The graphs are then fed to GNN frameworks which perform graph learning processes. Therefore, we can apply graph learning tasks to models as they are represented as graphs.

The paper is structured as follows. In Section 2 we briefly describe existing tools that are used by our encoder. Section 3 presents a running case that is used in Section 4 to describe our encoding process. Section 5 illustrates the current variability encoded in our DSL. Section 6 evaluates an application of the encoder. Section 7 outlines the main related work, and Section 8 concludes the paper.

2. Background

In this section, we briefly describe PyEcore and NetworkX, two existing tools that our encoder leverages to access models and produce graphs.

2.1. PyEcore

PyEcore is a Python implementation of the Eclipse Modeling Framework (EMF) [8]. It is open-source and is very similar to the Java implementation in terms of functionality. However, there are some differences. For example, there are no factories in PyEcore, but, it is possible to implement them by some tricks. The PyEcore library allows us to load and register existing metamodels (in XMI format, and conforming to Ecore) and different models (in XMI format) and makes it possible to work with their elements. A model is loaded in memory either from a resource (XMI, JSON) or created programmatically with PyEcore statements. Then, it is possible to navigate the model from one point to another using the meta-attribute/reference names. The attribute and reference values are accessible in the same way.

2.2. NetworkX

NetworkX is an open-source Python package that provides graph creation, manipulation, and visualization [9]. There are also functions to compute statistics in the network such as the number of connected elements, and its diameter. NetworkX provides a function for visualizing the graphs with nodes and edges. There are class methods that allow the manipulation of the network such as adding or removing nodes and edges. In NetworkX, it is possible to attach some attributes such as weight, color, name, or any other needed attribute to each graph, node, or edge. A key/value pair for each graph, node, or edge is held in a dictionary, and they are changeable using the corresponding graph functions.

3. Running Example

The Family metamodel in Fig 1 is considered as a running example. The *Family* class has one attribute *Surname* and two references with *Member* and *Address* classes. The Member class has three attributes *firstName*, *age*, and *education*. The education attribute just accepts values of *Edu* enumeration. The address class has three attributes *street*, *alley*, and *number*. We have generated some models conforming to this *Family* metamodel. One of the instances is shown in Fig 2. The *Lee Family* object has three *Member* objects and one Address object. Now consider that we want to do a learning task on the models. We may want to feed the model to PyTorch. However, there is no possibility to load models directly in PyTorch. Graph learning frameworks such as PyTorch Geometric and DGL propose deep learning on graphs. Models are similar to graphs, but they are not the same. So, we want to generate a graph that is 1) able to be fed to graph learning frameworks, 2) optimized for deep learning and 3) a representation of the source models. In the next section, we present our proposed encoding process.

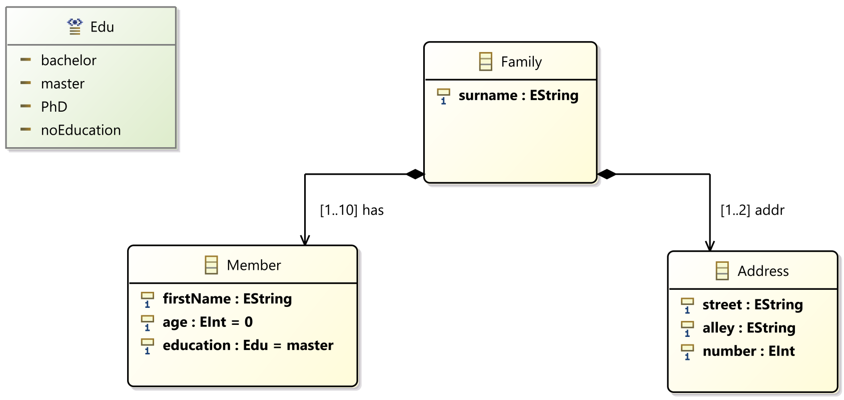

Figure 1: The Family metamodel

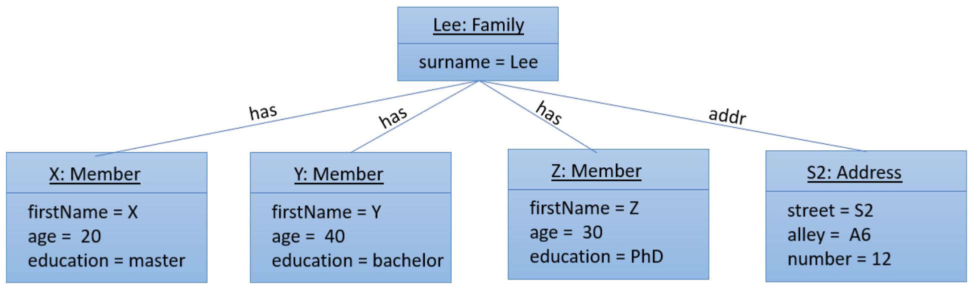

Figure 2: An instance model of Family metamodel

4. Encoding Process

The architecture of our proposed encoding of models is illustrated in Fig 3. It consists of two main phases: loading the models in PyEcore to navigate in the model and creating the corresponding graph using the NetworkX library. In the first phase, the model is imported to PyEcore. In general, both the model and the corresponding metamodel should be imported. However, the process can also be used to encode metamodels. In this case loading the metamodel alone is sufficient. We propose to use a DSL based on the YAML syntax to customize the encoding according to user preferences. Then the model is fed to a function to navigate in the model, get the model elements and create the corresponding graph. In this study, we concentrated on models with a tree structure, beginning the traversal from the root element. The root attributes are extracted in PyEcore. For each element, a node is created in the graph using NetworkX methods and the related attributes are assigned to it. When the first node is created, the elements that have references from that node are extracted, and for each of them, the same procedure of creating nodes and assigning attributes is performed, recursively. For each reference between two elements, an edge is created in the graph. This process is repeated until no other elements are left.

The implementation of our DSL makes use of the aforementioned PyEcore and NetworkX libraries. Our proposal for the DSL syntax is to use a YAML file. If a YAML file exists, the encoding is done by taking into account the priorities specified by the user. For example, it is possible to exclude a specific attribute from participating in the coding and being present in the final graph. The structure of the YAML file is explained in section 5 in more detail.

In the first phase, the metamodel (Ecore file) and the model (XMI file) are loaded into PyEcore resources. Then, the working directory is searched for an existing YAML configuration file. If the file exists, the attributes of all objects are extracted according to the preferences specified in the YAML file, otherwise, all the attributes in all the classes are extracted and saved in a dictionary named *attributeNames*. The class name of each attribute is also saved in the dictionary: *attributeNames[classname][attributename]*. This dictionary is used as the attribute features of all the nodes in the output graph, because the graph is homogeneous and all the nodes have the same type and the same attributes. The *attributeName* for the running example is presented in Listing 1.

The root of the model is then passed to a function that creates the corresponding graph for

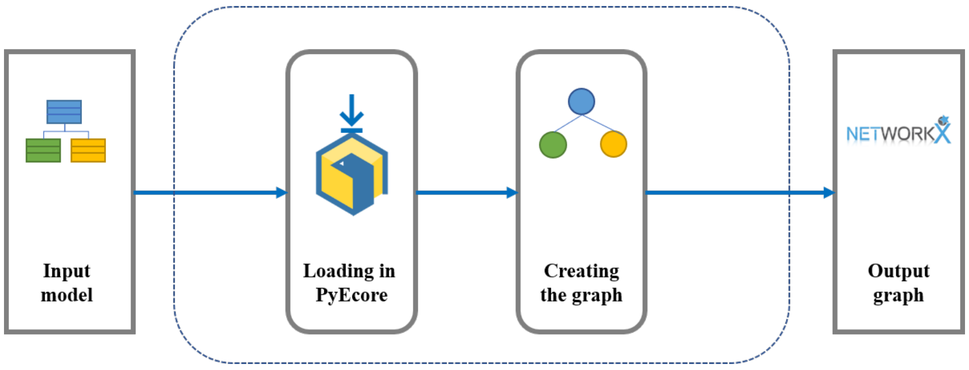

Figure 3: The encoding architecture

```
attributeName: {
   'Family': {'surname': 0},
   'Member': {'firstName': 0, 'age': 0, 'education': 0},
   'Address': {'street': 0, 'alley': 0, 'number': 0}}
```

Listing 1: Attribute names of the model in Fig 2

```
[(1, {'surname': 1, 'firstName': 0, 'age': 0, 'education': 0,
'street': 0, 'alley': 0, 'number': 0}),
(2, {'surname': 0, 'firstName': 0, 'age': 0, 'education': 0,
'street': 1, 'alley': 1, 'number': 1}), (3, {'surname': 0,
'firstName': 1, 'age': 1,'education': 1, 'street': 0,
'alley': 0,'number': 0}), (4, {'surname': 0, 'firstName': 1,
'age': 1, 'education': 1, 'street': 0, 'alley': 0,'number': 0}),
(5, {'surname': 0, 'firstName': 1,
'age': 1, 'education': 1, 'street': 0, 'alley': 0,'number': 0})]
```

Listing 2: node attributes regarding the existence of each attribute in each node

the model. At this point, a node is created in the graph for the root object, and the attribute list is initialized. For this purpose, an attribute is assigned to the node for each value in the *AttributeNames* dictionary, with the value equal to zero. Then, the actual attributes that exist in the element are searched and the corresponding attribute is set with the actual value. If the attribute values do not matter, it is possible to use 0 and 1 for the attribute values, just to distinguish if the attribute is set or not in the node. Listing 2 shows the node attributes for the model in Fig 2. For example, the first node *(Lee: Family)* has just the *Surname* attribute, and just the surname is set to 1 and other attribute values remain 0.

After creating the first node and assigning the attributes, the nodes that have references with

```
[(1, 2, {'label': 'addr'}), (1, 3, {'label': 'has'}),
(1, 4, {'label': 'has'}), (1, 5, {'label': 'has'})]
```

Listing 3: The edges of the graph representing the model in Fig 2

it are searched. The *Lee: Family* element in Figure 2 has four references, therefore, a node is created in the graph for the first object (e.g. X: Member) and an edge is established between *Lee: Family* and *X: Member.* Each edge has an attribute named *label* that specifies the name of that reference. The edge labels of the model is shown in Listing 3. In this case, the label of the edge between *Lee* and *X* would be *"has".* Then the attributes of *X: Member* are assigned and again it is searched for the references of *X*. As *X* has no references with any other objects, the next object *(Y: Member)* is examined and a reference with a label *has* is created between *Lee: Family* and *Y: Member.* This procedure is continued until no other element is left in the model and the graph will then be a complete representation of the model.

Now, we have represented the model in the graph world in the NetworkX format. If there are several models conforming to the same metamodel, all can be translated to their corresponding graph. This way we have a dataset of models used as an input dataset for a graph learning program such as classification, clustering, etc. In the following section, we present the YAML configuration file used to customize the encoding according to user preferences.

5. Encoding Configuration DSL

We propose a language to customize the encoding according to user preferences, since we believe that the user knows the domain the best, and the encoding would be more efficient by considering user knowledge. For example, the user may know that the value of a given attribute does not matter for a specific learning task. In this case, it is possible to specify this using our DSL, and it would simply be taken into account in the encoding. The YAML configuration file allows the user to specify what he has in mind to have better and more effective encoding. YAML [10] is a recursive acronym for "YAML Ain't Markup Language". It is a data serialization language, which helps to work with data. It contains some sections including structural information and raw data. It is relatively easy to use and compatible with most programming languages, such as Python. The class diagram of the proposed DSL is illustrated in Fig 4, and a YAML structure for the model in Fig 2 is shown in Listing 4.

The white classes in Fig 4 are references to classes imported from Ecore (as denoted by `ecore::` prefixing their names), and are not part of the language. The proposed structure allows the user to target each metamodel, class, attribute, or reference hierarchically, and to apply encoding preferences using the provided keywords. The metamodel is specified with its URI. *includeAllAttributes*/*excludeAllAttributes* in *packageEncoding* and *classEncoding* enable the user to include or exclude all of the attributes in a package's classes or in a specific class. The default value of each attribute is shown in the class diagram. It is not necessary to set all attributes in the YAML file, and omitting to write an option simply means that the default value is desired.

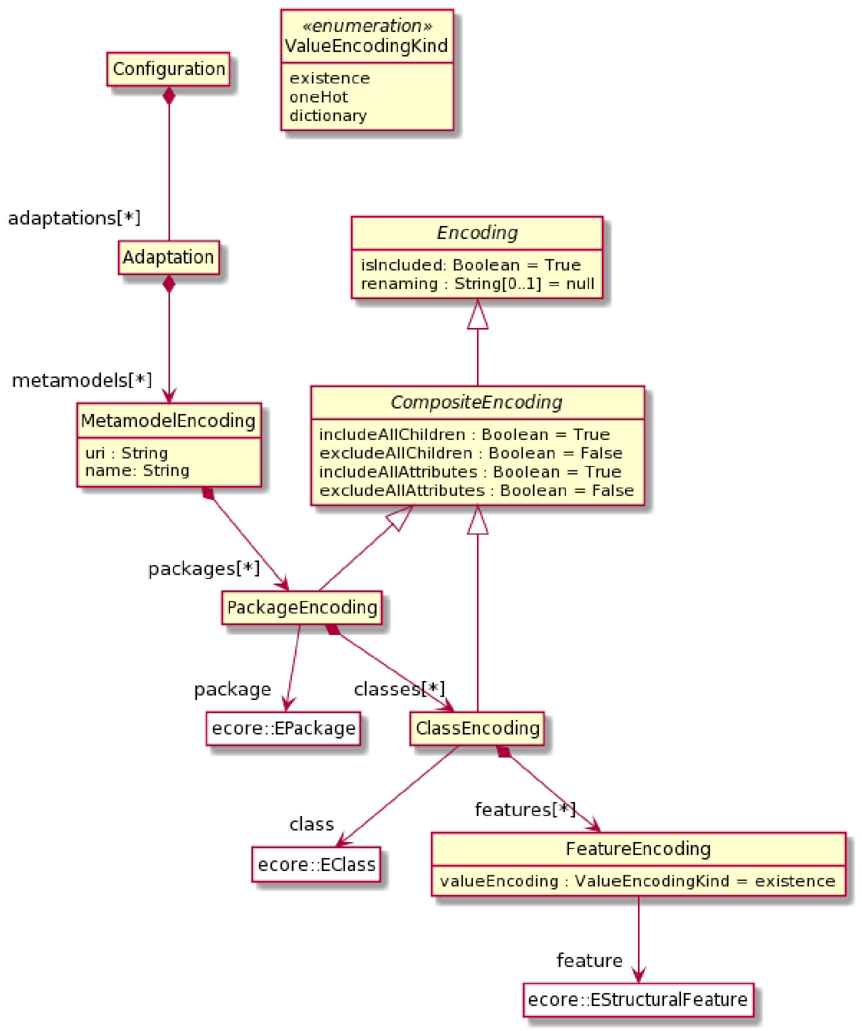

Figure 4: DSL class diagram

If both *includeAllAttributes* and *excludeAllAttributes* are set to the same values, the value for *includeAllAttributes* is just considered and a warning message is issued to alert the user. There are three encoding options: *isIncluded*, *renaming*, and *encoding*. The first one specifies whether the class or feature should be present in the encoding or not. As already mentioned, the options are hierarchically structured and the inner sections have more priority than the outer ones. For example if the *excludeAllAttributes* of a class is set to True, but *"isIncluded"* for a specific attribute of that class is set to True, it will be included in the encoding process. As depicted in Listing 4, in this example, the user prefers not to have the attribute *"surname"* in the encoding, so the value of *"isIncluded"* is set to False. The next encoding option is *"renaming"*, which allows the user to change a class or an attribute name. For example, there are two attributes in two

```
adaptations:
    metamodels:
        packages:
            Family:
                    uri:www.Family.com
                    classes:
                        Family:
                            isIncluded: true
                            includeAllChildren: true
                            features:
                                surname:
                                    isIncluded: false
                                    renaming: lastname
                                has:
                                    isIncluded: true
                        Member:
                            features:
                                education:
                                    valueEncoding: dictionary
```

Listing 4: A YAML file example for the model in Fig 2

different classes named *"surname"* and *"lastname"*, the user knows that they are the same, and he renames one of them to the other in order to have the same feature space in the encoding. *"valueEncoding"* is the last one. It can be used to specify how an attribute value should be encoded. There are three possible options. *"existence"* just specifies if an optional attribute has a value or not, *"oneHot"* translates the value to a onehot encoding, and *"dictionary"* uses a dictionary to have a distinct natural number represent each distinct value. For example in Listing 4, the value of *valueEncoding* for attribute *"Education"* is set to *"dictionary"*, because there are just three values for that and it is better to have numbers 1 to 3 representing each value.

6. Evaluation

We evaluate our work by choosing an existing graph dataset and creating a metamodel representing all the graphs in the dataset. Then we choose some graphs and generate corresponding models, which completely represent the graphs. The models are then fed to the encoding program and they will be encoded in the NetworkX format. Finally, we compare the result with the original ones in NetworkX format. The purpose of this process is to show that we can achieve the same encoding as the one used in the original dataset.

For this purpose, we choose the AIDS dataset from the TUDataset [11], because it is a standard dataset used in the literature, and it also contains node labels, edge labels, and node attributes, which are similar to what we have in models. AIDS is a collection of antivirus screen chemical compounds, and contains 2000 graphs. Looking into this dataset, each node has four attributes *chem, charge, x, and y*, and a label indicating the *symbol* of that specific node. There are 38 types of symbols, each converted to an integer value and assigned as the label of nodes. There are also three types of edges between the nodes converted to an integer number and assigned as the edge label between two nodes.

The metamodel of the AIDS graph dataset which we defined is depicted in Fig 5, according to the information of the dataset and our understanding of the graph instances. We considered *"symbol"* as an attribute of the class in the metamodel, but then we rename it to *"label"* in the DSL to indicate the node label. We choose the third graph of the dataset and generate the model representing the graph. This graph has nine nodes and eight edges between nodes. The model of the graph is shown in Fig 6. In this graph dataset, the attribute names do not matter, and only their values are stored in a vector to be considered in the learning process. Therefore, we ignore the names of the attributes and create an attribute vector named *"attributes"* that contains the values of the attributes in each node.

To evaluate the model and the encoding, we converted the examined graph to network format using the *"tud_to_networkx"* function proposed in [11] and printed the nodes and edges data before comparing the original data encoding with ours. The printed data of both are illustrated in Listing 5. They are quite similar, considering that the numbering order of nodes and the order of features in the two views are only different because the model traversing strategy is different in our method.

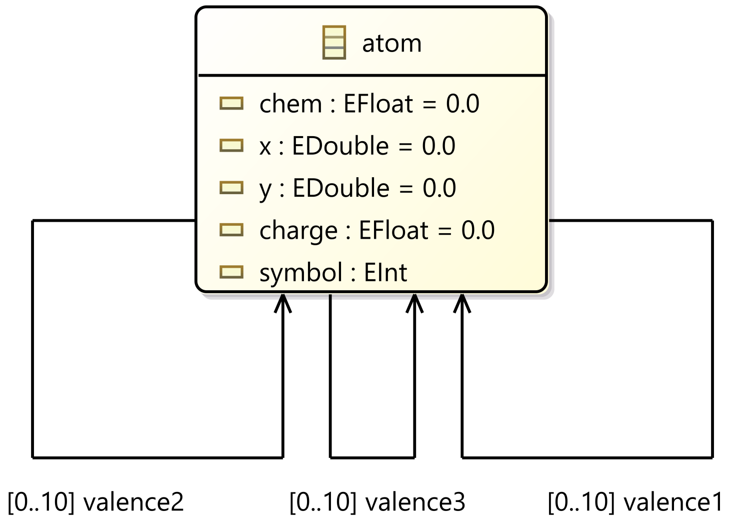

Figure 5: The metamodel of the AIDS graph dataset

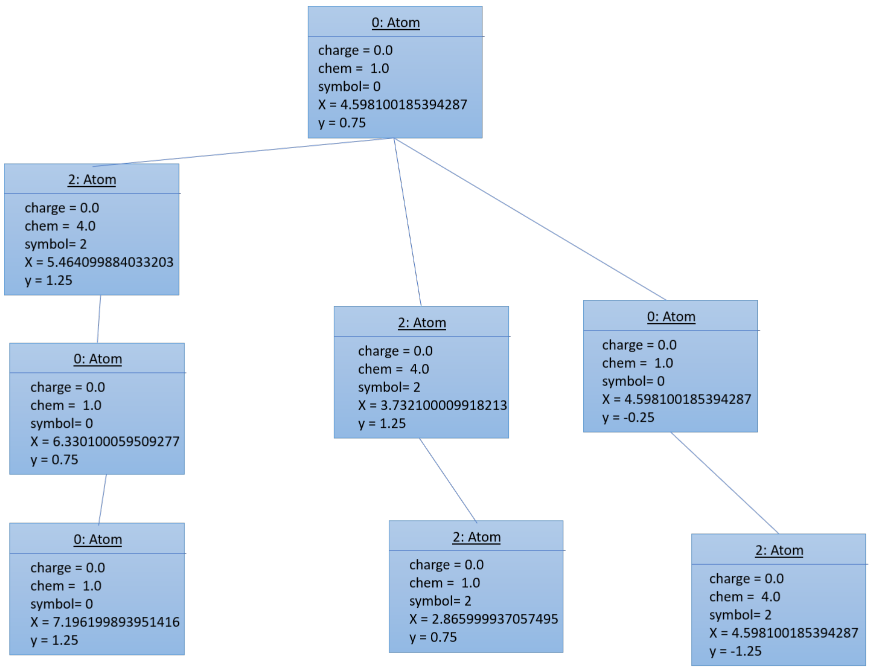

Figure 6: The model representing the third graph of the AIDS dataset

7. Related Work

There are some attempts in the literature to apply DL on models, each of which propose an encoding technique to feed the model to DL frameworks. Nguyen et al. [12] introduce classification of metamodels by a tool implementing a feed-forward neural network. They propose some preprocessing tasks to transform a metamodel into a feature vector. The tasks consist of a term extractor that parses the terms in the metamodel, and then of an NLP Normalizer that normalizes the extracted terms by performing some Natural Language Processing tasks. Burgueno et al. [13] propose a neural network architecture based on Long Short-Term Memory (LSTM) Neural Networks to perform model transformation automatically. They propose to apply a preprocessing method to represent models as trees before feeding them to an Artificial Neural Network (ANN). After transforming the model into a tree using the proposed strategy, a normalization process is applied to overcome the limitations in ANNs. Couto et al. [14] propose to classify unstructured models to metamodels using Multi Layer Perceptrons (MLP). They translate a metamodel containing classes, attributes and references into a set of name/value pairs in a JSON format. Then a binary vector is created as the MLP's input feature.

Each of these works focuses on a specific learning task, and encodes models according to its specific needs. By contrast, we do not focus on any specific task, but rather propose a general encoding approach that aims at being applicable to any learning task. The contribution of our work is leveraging graph learning frameworks and representing models as graphs, so that we can leverage their capabilities in graph learning processes. The most distinguishing feature of our approach is that it makes it relatively easy for users to specify their encoding preferences. All of the existing works apply a rough encoding process to the models and the encoding output is completely bonded to the input model.

8. Conclusion and Future Work

We proposed a DSL for encoding models as graphs to be able to feed models to deep learning frameworks. We leveraged two Python libraries: PyEcore and NetworkX, in order to extract the elements and data from the models and create graphs according to the extracted elements. A YAML file is used to apply user knowledge and preferences to the encoding process to make the encoding more efficient. Initial experimental results show that our encoding is able to convey all the data in the model to the graph and maintain the structure as well.

Our DSL supports several types of value encoding, including several binary formats for strings. In future work, we plan to design heuristics for determining an appropriate encoding from structural analysis of the metamodel and characterization of the learning objective. Besides, more evaluation will be necessary to confirm that our approach also works with models conforming to more complex metamodels.

References

[1] D. C. Schmidt, Model-driven engineering, Computer-IEEE Computer Society- 39 (2006) 25.

[2] T. Stahl, M. Völter, K. Czarnecki, Model-driven software development: technology, engineering, management, John Wiley & Sons, Inc., 2006.

[3] M. Fey, J. E. Lenssen, Fast graph representation learning with pytorch geometric, arXiv preprint arXiv:1903.02428 (2019).

[4] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, et al., Deep graph library: Towards efficient and scalable deep learning on graphs. (2019).

[5] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al., Relational inductive biases, deep learning, and graph networks, arXiv preprint arXiv:1806.01261 (2018).

[6] L. Backstrom, J. Leskovec, Supervised random walks: predicting and recommending links in social networks, in: Proceedings of the fourth ACM international conference on Web search and data mining, 2011, pp. 635–644.

[7] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, R. P. Adams, Convolutional networks on graphs for learning molecular fingerprints, arXiv preprint arXiv:1509.09292 (2015).

[8] V. Aranega, Pyecore: A python (ic) implementation of the eclipse modeling framework, modeling-languages. com (2017).

[9] A. Hagberg, P. Swart, D. S Chult, Exploring network structure, dynamics, and function using NetworkX, Technical Report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[10] O. Ben-Kiki, C. Evans, B. Ingerson, Yaml ain't markup language (yaml™) version 1.1, Working Draft 2008-05 11 (2009).

[11] C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, M. Neumann, Tudataset: A collection of benchmark datasets for learning with graphs, arXiv preprint arXiv:2007.08663 (2020).

[12] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, A. Pierantonio, L. Iovino, Automated classification of metamodel repositories: A machine learning approach. in 2019 acm/ieee 22nd international conference on model driven engineering languages and systems (models), 2019.

[13] L. Burgueño, J. Cabot, S. Gérard, An lstm-based neural network architecture for model transformations, in: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), IEEE, 2019, pp. 294–299.

[14] W. O. Couto, E. C. Morais, M. D. Del Fabro, Classifying unstructured models into meta-models using multi layer perceptrons., in: MODELSWARD, 2020, pp. 271–278.

```
Graph nodes = [(1, {'attributes': [1.0, 4.598100185394287,
    0.75, 0.0], 'labels': [0]}), (2, {'attributes': [1.0,
    4.598100185394287, -0.25, 0.0], 'labels': [0]}), (3,
    {'attributes': [4.0, 4.598100185394287, -1.25, 0.0],
    'labels': [2]}), (4, {'attributes': [4.0,
    3.732100009918213, 1.25, 0.0], 'labels': [2]}), (5,
    {'attributes': [1.0, 2.865999937057495, 0.75, 0.0],
    'labels': [0]}), (6, {'attributes': [1.0, 2.0, 1.25, 0.0],
    'labels': [0]}), (7, {'attributes': [4.0,
    5.464099884033203, 1.25, 0.0], 'labels': [2]}), (8,
    {'attributes': [1.0, 6.330100059509277, 0.75, 0.0],
    'labels': [0]}), (9, {'attributes': [1.0,
    7.196199893951416, 1.25, 0.0], 'labels': [0]})]
Graph edges = [(1, 2, {'labels': 1}), (1, 4, {'labels': 1}),
    (1, 7, {'labels': 3}), (2, 3, {'labels': 2}), (4, 5,
    {'labels': 1}), (5, 6, {'labels': 1}), (7, 8, {'labels':
    1}), (8, 9, {'labels': 1})]
--------------------------------------------------

Graph nodes = [(0, {'labels': [0], 'attributes': [1.0, 0.0,
    4.598100185394287, 0.75]}), (1, {'labels': [0],
    'attributes': [1.0, 0.0, 4.598100185394287, -0.25]}), (2,
    {'labels': [2], 'attributes': [4.0, 0.0, 4.598100185394287,
    -1.25]}), (3, {'labels': [2], 'attributes': [4.0, 0.0,
    5.464099884033203, 1.25]}), (4, {'labels': [0],
    'attributes': [1.0, 0.0, 6.330100059509277, 0.75]}), (5,
    {'labels': [0], 'attributes': [1.0, 0.0, 7.196199893951416,
    1.25]}), (6, {'labels': [2], 'attributes': [4.0, 0.0,
    3.732100009918213, 1.25]}), (7, {'labels': [0],
    'attributes': [1.0, 0.0, 2.865999937057495, 0.75]}), (8,
    {'labels': [0], 'attributes': [1.0, 0.0, 2.0, 1.25]})]

Graph edges = [(0, 1, {'labels': [0]}), (0, 3, {'labels':
    [1]}), (0, 6, {'labels': [0]}), (1, 2, {'labels': [2]}),
    (3, 4, {'labels': [0]}), (4, 5, {'labels': [0]}), (6, 7,
    {'labels': [0]}), (7, 8, {'labels': [0]})]
```

Listing 5: The nodes and the edges of the original graph and our encoded graph