IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 14 September 2023, accepted 2 October 2023, date of publication 10 October 2023, date of current version 25 October 2023.

*Digital Object Identifier 10.1109/ACCESS.2023.3323591*

RESEARCH ARTICLE

# Link Prediction for Completing Graphical Software Models Using Neural Networks

**ONUR LEBLEBICI[1], TUGKAN TUGLULAR[1], (Member, IEEE), AND FEVZI BELLI[1,2], (Member, IEEE)**

[1]Department of Computer Engineering, İzmir Institute of Technology, 35430 İzmir, Turkey
[2]Department of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, 33098 Paderborn, Germany

Corresponding author: Tugkan Tuglular (tugkantuglular@iyte.edu.tr)

**ABSTRACT** Deficiencies and inconsistencies introduced during the modeling of software systems may result in high costs and negatively impact the quality of all developments performed using these models. Therefore, developing more accurate models will aid software architects in developing software systems that match and exceed expectations. This paper proposes a graph neural network (GNN) method for predicting missing connections, or links, in graphical models, which are widely employed in modeling software systems. The proposed method utilizes graphs as allegedly incomplete, primitive graphical models of the system under consideration (SUC) as input and proposes links between its elements through the following steps: (i) transform the models into graph-structured data and extract features from the nodes, (ii) train the GNN model, and (iii) evaluate the performance of the trained model. Two GNN models based on SEAL and DeepLinker are evaluated using three performance metrics, namely cross-entropy loss, area under curve, and accuracy. Event sequence graphs (ESGs) are used as an example of applying the approach to an event-based behavioral modeling technique. Examining the results of experiments conducted on various datasets and variations of GNN reveals that missing connections between events in an ESG can be predicted even with relatively small datasets generated from ESG models.

**INDEX TERMS** Event-based modeling, graph neural networks, link prediction.

## I. INTRODUCTION

In the analysis and design phase of software development, a thorough understanding of user requirements is crucial [1], [2]. The models developed during the phase of analysis and design impact the whole software development lifecycle [3], [4]. The degree of alignment between business process models and software system models should be high [5]. Engineers may be unable to reduce this complexity using conventional software modeling techniques. Therefore, it is important to predict and recommend interactions between software components, such as classes, events, and user interactions (UI) elements, in modeling. From the software engineering perspective, deciding how to make component and user interactions is error-prone and, therefore, requires considerable effort, as these interactions may semantically represent relationships, such as "follows" [6]. Instead of placing the entire burden on the software engineer, engineers can be assisted in modeling [7], [8], so that the interaction among the components, the user, and the system in a software system can be modeled with some recommendations. In this paper, when we mention software models, we mean interactive or event-based software models, but not logical or structural software models.

There are many different models and tools used in software modeling [9], [10]. Most of these models are graph-based, and there is a well-established theory of graph transformations [11], which has several system modeling and software engineering applications [12], [13]. For instance, in the setting of co-evolution of models and meta-models, Taentzer et al. [14] defined co-evolution rules using graph transformations. By using a set of graph transformations, they made sure that models conformed to a meta-model and met the constraints that had been imposed on them. Another example is the Henshin toolset [15], which uses of graph transformations to support model migration and evolution.

The associate editor coordinating the review of this manuscript and approving it for publication was Giacomo Fiumara.

The motivation for this work is to help software engineers during interactive or event-based graphical modeling of the software under consideration. The proactive exploration for absent artifacts inside software models is a strategic endeavor aimed at safeguarding the quality, functionality, security, and maintainability of the software. The utilization of this approach aids in the optimization of the software development process, resulting in cost reduction and the eventual delivery of a software that is more resilient and dependable. The quality of a software system is directly related to how accurately it's been modeled. A complete and detailed model helps ensure that the final system meets the desired standards of functionality, performance, and security. Moreover, the early identification of missing elements can result in time and resource savings, hence mitigating the necessity for significant modifications or repairs during the latter stages of the development process.

The proposed approach should prevent or reduce missing links in these graphical models and quality of the models will be increased. Since these models are used for designing, coding, and testing in the software development processes later, any deficiencies and errors that may occur in this process can result in very high costs. Quality of modeling directly affects the quality of the software.

The missing links in the graphical models this paper considers may stem from impairments in the following situations and processes:

- Depending on the project size, several software development teams may work simultaneously. They should manage this kind of variety to create complete and consistent specifications. Deficiencies and inconsistencies in the modeling step can cause missing links in the graphical models.
- In addition to deficiencies and inconsistencies, ambiguities and unstated assumptions can also be the reason for missing links in the graphical models. Since requirements/specifications can be interpreted in multiple ways, ambiguities can cause the models to have missing links. Unstated assumptions are implicit expectations not mentioned in the requirements/specifications but assumed to be understood. If not understood, there can be missing links in the graphical models.
- Novel techniques support the design and modeling process. The GUI Ripping, for example, enables automatically creating a model of the GUI of an application under test from its executable binary code [16]. Such automatically generated models have to be handled with particular caution. Therefore, numerous attempts are necessary that deliver several models on a trial basis [17]. In this case, also, there will be a variety of impairments that should be managed to create complete and consistent specifications.

Missing links in the graphical models, such as missing requirements and specifications, signify the presence of ambiguities, inconsistencies, and incompleteness [18]. The origins of these issues can be attributed to miscommunication or misconceptions between stakeholders and developers. If developers or analysts lack familiarity with the particular domain, there is a risk of missing specifications. Large and complex software projects are more prone to omitting specific details. In dynamic environments characterized by frequent changes, it is possible for specific requirements/specifications to be overlooked or forgotten [19], [20], [21].

The prediction of missing links in graphical software models is an integral part of the process of discovering missing requirements and specifications. Commonly accepted methodologies for identifying these absent specifications including employing organized techniques to collect needs from relevant stakeholders, conducting thorough reviews and inspections of use cases or user stories, examining models and diagrams, and utilizing prototyping techniques [1], [22]. These procedures are carried out manually rather than being automated [23], [24].

This paper is based on the thesis titled "Application of Graph Neural Networks on Software Modeling" [25]. The proposed approach selects modeling graphical user interface (GUI) for an application-oriented representation and discusses how to compensate for deficiencies in a GUI model. These deficiencies can be considered as the mutants of the original graphical software models [26], where mutations are defined as minor modifications to these models, such as edge removal [27]. Mutation analysis is used to evaluate the effectiveness of test suites [26], not to compensate for deficiencies in a GUI model. Therefore, mutation analysis differentiates from link prediction.

Most of today's software applications use a graphical user interface (GUI) as a front end to interact with the user and other systems. In GUI software, interface components form the visible GUI structure, and these components accept sequences of user events, for example, mouse clicks and type-in-text, that alter the state of the software. Thus, software graphical user interfaces (GUIs) can be modeled as sequences of events of the GUI components [28], [29].

The graph-based modeling technique considered in this work is Event Sequence Graphs (ESGs) [28]. The formal structure of ESG, representing a directed graph, allows us to use the terms and notions of Graph Theory and exploit its results developed over many centuries. Event-based graphical techniques serve as a prevalent approach for behavioral modeling. An event, as an externally observable occurrence, such as a user's input or a system response, offers a lens into distinct stages of the system under consideration's (SUC) activity.

Among the most widely recognized event-based graphical techniques are those centered on event sequence graphs (ESGs) [28] and event flow graphs (EFGs) [30]. Alternatively, state-based methodologies can be harnessed for behavioral modeling, including finite-state automata (FSA) [31] or statecharts [32], among others. Viewed as an FSA, an ESG merges inputs and states into events, resulting in a one-sorted graph with a singular type of element

(circles). This can be seen as a simplification of a finite state automaton's state transition diagram (STD), based on [33]. Figure 1 visualizes both cases.

In ESGs, nodes (circles) symbolize events that define user actions and system behavior, while arcs signify sequences of these events. Consequently, ESGs direct their attention towards events, bypassing explicit state processing. This minimalist approach streamlines ESG learning and utilization, enabling designers to sidestep errors in their models. Importantly, a grasp of automata theory is unnecessary. Additionally, owing to its directed graph nature, ESGs lend themselves to efficient graph theory algorithms for analysis, validation, and optimization [34]. Benefiting from its foundation in automata theory, ESGs leverage the strengths of both theories.

Moreover, ESG modeling facilitates negative testing through complementation. By adding missing edges that denote illegal user-system interactions, ESGs simplify the identification of unexpected or undefined system responses. In contrast, EFGs and Statecharts are multi-sorted, encompassing various graphical elements with distinct semantics. This diversity hampers the direct application of graph theory outcomes and the concept of complementation.

ESGs are a well-established formal graphical model employed to represent the interactions inside a software system. However, it is not uncommon for these graphical models to exhibit missing links. Consequently, there arises a necessity for the development of a link prediction technique specifically tailored for graphical software models. The objective of this study is to examine and provide solutions for the research problems outlined below:

1) Apply, extend, if necessary, two state-of-the-art machine/deep learning-based link prediction approaches to ESGs. The objective of this extension is to make these approaches applicable to ESGs.
2) Evaluate the efficacy of both methods by using the existing ESG models.

Accordingly, this paper introduces an application of two graph neural networks (GNNs), which predict missing links between events defined in an ESG. For an ESG, a link means a transition between two events. Experiments were performed on four different datasets with two different customized GNN models, namely Seal-ESG and DeepLniker-ESG, to predict links that have not been existed before. The steps of the process to find missing links between ESG nodes are as follows:

1) Transform ESG models into graph-structured data and extract features of the nodes,
2) Train the GNN model,
3) Evaluate the performance of the trained model.

The results of the experiments show that the two customized GNN models can make recommendations on missing links or edges of the graph-based system models.

Our contributions are summarized as follows.

1) We present an application of GNNs in aiding graphical software modeling by predicting missing links.
2) We extend and customize two GNN models for the above mentioned application and compare their performances.

The outline of the paper is as follows. Section II provides essential information about the terms and terminologies used. Section III gives an overview of preliminary research on link prediction using GNN. Section IV introduces and explains the steps of the proposed approach, while Section V presents the evaluations of different datasets and GNN models using the proposed method. Section VI explores related work, and the last section provides conclusions and possible future work.

## II. EVENT SEQUENCE GRAPHS

Event sequence graphs (ESGs) are used for modeling system behavior [28]. ESGs might also be used to represent both the planned (i.e., proper) and undesired (i.e., exceptional) behavior of the system from the user's perspective [28]. Using discrete event-based models, ESGs concentrate on the externally visible behavior of computer-based systems [35]. ESGs model the interconnections between user events, environmental activities, and system reactions using an event-based structure [35]. The whole collection of interactions is derived from a series of ESGs, each representing a potentially endless number of event sequences [35]. This collection of event sequences is used to evaluate a computer system's intended and unintended behavior. The following event sequence graph definitions are utilized throughout this investigation. Figure 2 is an illustration of an event sequence graph. The below definitions of even sequence graphs are taken from [28] and [35].

*Definition 1:* An event sequence graph (ESG) is a directed graph where $V = \emptyset$ is a finite set of nodes (vertices) and $E \subseteq VxV$ is a finite set of *arcs (edges)* and $\Xi, \Gamma \subseteq V$ finite sets of distinguished vertices with $\xi \in \Xi, \gamma \in \Gamma$ called entry nodes and exit nodes, respectively [28].

The entry and exit vertices of an ESG are marked by applying the following convention: all $\xi \in \Xi$ are preceded by a pseudo vertex '[' $\notin V$ and all $\gamma \in \Gamma$ are followed by another pseudo vertex ']' $\notin V$ [28]. The entry and exit vertices which are demonstrated by '[' and ']' respectively, are called pseudo vertices and they are not included in $V$ [28]. The pseudo vertices are not included also in event sequences.

For the ESG given in Figure 2, the event set $V =$ get balance, select deposit, enter deposit amount, put money, select withdraw, enter withdraw amount, take money, the set of entry events $\Xi =$ get balance, select deposit, select withdraw, the set of exit events $\Gamma =$ get balance, put money, take money and the edge set $E =$ (get balance, select deposit), (select deposit, enter deposit amount), (enter deposit amount, put money), (get balance, select withdraw), (select withdraw, enter withdraw amount), (enter withdraw amount, take money), (put money, get balance), (take money, get balance). $E$ does not contain the edges from pseudo vertex '[', and to pseudo vertex ']'.

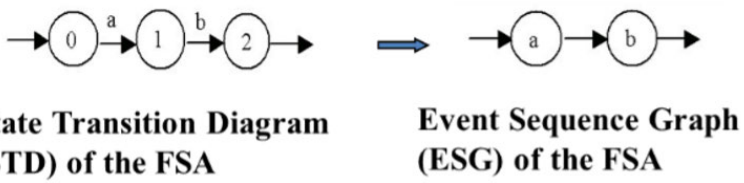


**FIGURE 1.** Finite-state automata as STD and ESG.

*Definition 2:* Let $(V, E)$ be an *ESG*. Then a sequence of vertices $< v_0, \ldots, v_k >$ is called an *event sequence (ES)* if the sequence is a walk on *ESG* [28].

Each edge of an ESG represents a legal event pair, or simply, an *event pair (EP)*. ES $< v_i, v_k >$ of length 2 is an EP [28]. select deposit - enter deposit amount - put money is an ES of length 3 of the ESG in which given in Figure 2.

*Definition 3:* An ES $< v_0, \ldots, v_k >$ is called a *complete event sequence (CES)*, if $v_0 = \xi \in \Xi$ is the entry and $v_k = \gamma$ is the exit. A CES represents a *test sequence* [35].

A CES is also a test sequence, i.e., test case, of the ESG and it is of the form ''(initial) user inputs → (interim) system responses → ... → (final) system response'' [6]. The ESG that is demonstrated in Figure 2, has a CES get balance - select withdraw - enter withdraw amount - take money which represents a walk from the entry of the ESG to its exit.

## III. GRAPH NEURAL NETWORKS FOR LINK PREDICTION

Graph Neural Networks, also known as GNNs, are a subset of neural networks developed to handle data organized in the form of graphs. Scarselli et al. [37] presented the concept of GNN, which adapts neural networks for graphs. They enhanced the Recurrent Neural Network (RNN) to apply to different kinds of graphs, including directed and undirected graphs, as well as cyclic and acyclic ones. Nevertheless, their approach is only valid for static graphs; it is not applicable to dynamic graphs. The strategy that has been suggested works for each vertex, feeding the knowledge of adjacent vertices into the recurrent neural network in a sequential fashion and repeating this procedure until the model becomes steady.

The general purpose of graph neural networks is to solve classification and regression problems of a graph that have not been encountered before with a pre-trained model. Graphs serve as a means of representing and illustrating the connections and associations that exist between various entities. In the context of a graphical software model, it is common for nodes to symbolize events, while edges are often used to denote links or interactions between these events. GNNs have an ability to efficiently capture and leverage complex networks' intricate relationships and associated information. A fundamental objective of GNNs involves acquiring significant representations, also known as embeddings, for the nodes inside a graph. Subsequently, these representations can be employed for various tasks such as node classification, i.e., assigning labels to nodes in a graph, and link prediction, i.e., predicting the likelihood of a link (or edge) formation between two nodes.

Message transfer is a fundamental concept in graph neural networks. Each vertex in a graph communicates its state to its adjacent vertices. At each iteration, neighboring state information is passed to a function, either a sum or an average, which modifies the vertex's state information. A vertex's hop count indicates how many vertices it must communicate. Initial research utilized a technique where every vertex repeatedly broadcasts its state to its neighbors. This repeated broadcast aims to accomplish maximum stability. However, significant computational overhead was incurred, particularly in big graphs, without achieving the desired accuracy.

The popularity of convolutional neural networks (CNNs) [38] has experienced significant growth in recent years due to its superior effectiveness compared to other neural network approaches. CNNs process input data through filters and subsequently downsample the results. The process of sampling can be accomplished by the utilization of functions such as average, minimum, or maximum. The application of these filters to the vertices results in the generation of sub-graphs for each individual vertex.

There are two types of graph convolutional neural networks: spectral models that employ graph Fourier base generalization [39] and spatial models that rely on message passing [37]. RecGNN's [37] message-passing technique is combined with convolution in spatial-based models. Weights are easily exchanged between locations and structures thanks to the local nature of the graph convolutions performed by these models. Spatial models are better than spectral models not only in terms of performance but also in terms of efficiency.

Zhang et al. [40] introduced the sort pooling technique, which enables classical CNNs to work on graphs, and called it the Deep Graph Convolutional Neural Network (DGCNN). The original GCN has been substantially enhanced by GraphSAGE [41]. This enhancement reduced computation costs and enabled GCNs to work on dynamic graphs. GraphSAGE enhances the representation of a vertex by an aggregator function, which consumes a predefined number of neighbors' features, not all like in the original GCN. Further improvements came from FastGCN [42] through the sampling algorithm. It uses the crucial vertices as part of the sample set instead of randomly selecting some number of vertices. Also, a vertex's important function for the receptive

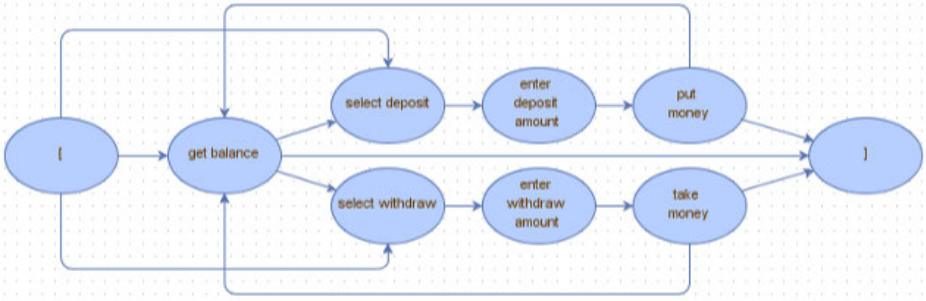


**FIGURE 2.** Simplified bank account ESG [36].

field is employed as a sample point across all layers rather than the neighbors of the vertex itself.

Veličković et al. [43] applied a self-attention approach to obtain the representation of a vertex through graph attention networks (GATs). The GAT architecture takes advantage of the model's multi-head attention to enhance its expressive power. The outcome is computed by averaging or adding the numbers calculated for each separate attention. In GAT's view, all attentions carry the same significance. Zhang et al. [44] improved this scheme by incorporating weights for each attention and called it the Gated Attention Network (GaAN) model. During aggregation, a GCN gives neighboring vertices explicit non-parametric weight, whereas a GAN learns the weights automatically through the neural network.

The process of link prediction attempts to determine whether or not there is a connection between the nodes that are defined on the graph [45]. Heuristic methods are currently utilized as one of the available strategies for link prediction. This approach might be useful in a few particular circumstances, but in general it has very poor performance. A hypothetical link between two nodes can be estimated, for instance, based on the number of neighbors that both nodes share. Although this method is successful in predicting social networks, it does not demonstrate any success when it comes to forecasting intra-molecular bonds [46].

The SEAL framework [47] was one of the earliest and most effective attempts to apply GCN to the link prediction problem. The SEAL framework uses DGCNN to learn the embedding features of sub-graphs extracted from networks of connected nodes. The obtained model is then utilized to predict links.

Gu et al. [48] modified Veličković et al.'s GAT model [43] for link prediction. The original GAT model requires a complete graph. Gu et al.'s mini-batch sampling-based DeepLinker [48] uses neighborhoods. DeepLinker uses the GraphSAGE architecture [41] with the exception of using GAT instead of GCN. Using the neighbors' attentions, DeepLinker generates a representation for each node.

## IV. PROPOSED METHOD

In this work, ESG models are considered as the specification or design of software systems, and software systems are developed or tested based on these. Creating better ESG models will help software engineers to build better software systems that meet user expectations. This work proposes a method that finds missing links in ESGs. As in all software modeling methods, including ESGs, the absent relationships between the software model's components naturally affect all software development processes.

Developing machine learning (ML) or deep learning (DL) models is a systematic process that involves multiple steps. The workflow we used in our study for developing ML/DL models is shown in Figure 3. In the Data Collection step, raw data is gathered raw data relevant to the problem. Publicly available datasets are usually favored. Data preprocessing and feature engineering are performed in the Data Transformation step to prepare data to feed to the selected models. A suitable ML/DL algorithm based on the problem type is chosen in the Model Training step, and an architecture is selected. Then, the training data is fed into the model. In the Evaluation of the Model Accuracy step, validation data is used to tune hyperparameters and prevent overfitting. In the last step, the model is validated on the unseen data. The ML/DL model's performance is monitored using loss, accuracy, and AUC metrics. Similar ML/DL development workflows exist in other application domains, such as in materials research [49] and in cardiovascular disease research [50].

The application of each step of the workflow depicted in Figure 3 to our problem is as follows. In the data collection phase, a bank account [36], email [36], student attendance [36], and reservation system [51] models are used. These models are drawn by the Test Suite Designer (TSD) tool employed in [51]. The TSD tool generates an XML file with a mxe extension. The proposed data transformation method reads a mxe file and transforms it to the graph data that graph neural network models need. During the training phase, GAT and GCN neural network models are used. Three performance metrics, cross-entropy loss, area under curve, and accuracy, are used to measure the performance of trained models. In the following sections, details of each phase are explained.

### A. DATA COLLECTION

One of the most challenging processes while working on neural networks is to find or create the data sets. For this


**FIGURE 3.** The workflow used in this study for developing machine/deep learning models.

**TABLE 1.** Graph data details of dataset models.

| Dataset | Num. of Nodes | Num. of Edges |
|---|---|---|
| ISELTA | 68 | 249 |
| Student Attendance System | 50 | 95 |
| Bank Account | 21 | 38 |
| Email | 19 | 35 |

purpose, previously prepared ESG models were used. The models, i.e., data sets, used in this work are listed below and the details of their graphical models are given in Table 1.

- Bank Account [36]: Operations on a bank account, such as withdrawal, viewing balance, depositing money, withdrawing money, and requesting interest, are modeled.
- Email [36]: Application about preparing new messages, viewing the mailbox, answering and forwarding messages, creating an address book, and creating auto response messages features are modeled.
- Student Attendance [36]: An attendance/nonattendance tracking application is modeled. In this model, there are two different roles as student and teacher. Students can enter and follow attendance information, and teachers can organize and monitor classes on a calendar.
- ISELTA [51]: It is a model of an application that allows users to edit and view their profiles, list hotels, and make reservations.

### B. DATA TRANSFORMATION

An essential part of the data transformation is embeddings [52]. Neural network embeddings are helpful because they can reduce categorical variables' dimensionality and meaningfully represent categories in the transformed space. This work transforms each graph node into its low-dimensional representations through node embeddings, which are used for neural network inputs.

Many software systems we encounter are complex structures with many details. Regardless of their experience level, people have an upper limit on their ability to analyze. Since the complexity of software systems makes it impossible to handle all aspects of the system by one person at once, such systems must be designed in parts. Each sub-part to be developed is dealt with and prepared separately by domain experts and software engineers. The models created as a result of these designs are relatively small. The graphs transformed from these models are naturally small. As the number of embedding vectors and elements within the embedding vectors increases, the node's representation space naturally grows. Therefore, the designer should carefully select the number of embedding vectors to represent small graphs properly. If the representation space to represent nodes on short networks expands, the representation will not change from high dimensionality to low dimensionality despite the use of embedding. One of the fundamental functions of embeddings is to transform a high-dimensional representation into a low-dimensional one. This representation space must be far less than the number of nodes in the graph.

Since ESGs are small graphs, it would be more appropriate to represent the nodes belonging to Event Sequence Graphs with a single embedding vector. In this context, the ''Event Type'' embedding vector is chosen as the most suitable for learning since the neural network best expresses the patterns between nodes. A simple mapping operation can transform from node names defined in an ESG to ''Event Type'' embedding elements. For these reasons, ''Event Type'' embedding is used in this work. Clearly, embeddings can be learned and reused in different models. However, embedding vectors are generated manually in this work.

Determining a node's feature, i.e., its ''Event Type'' embedding, is a manual process. It depends on the ESG under consideration because different domains may require different kinds of node names. The chosen names for the evaluation are listed in Table 2. They are determined as generic as possible for a regular application. The mapping table for converting node names to event types, or features, is listed in Table 2.

Files generated by TSD have a mxe extension and are formed in XML notation. ESGs need to be flattened so that they can be appropriately analyzed. A mxe model parser tool is built to help with this task. This tool has two Python functions implemented. The first function [53] flattens cascaded ESGs, which are ESGs with sub-ESGs. Once a flattened ESG has been obtained, the second function [53] parses the mxe file to get the edges and vertices of the flattened ESG.

For each mxe input file, the mxe parser application generates three output files: nodes, edges, and mappings. The node output file is tab-separated, and each line represents a node and features of the node except the first line. The first line represents the number of nodes and the number of node features this graph contains. The edge output file is also tab-separated, and the first line represents the number of edges and the number of edge features defined for this graph. Other lines are structured as follows, the first column is the source

**TABLE 2.** Event type to node name mappings.

| Id | Event Type | Matching Node Names |
|---|---|---|
| 0 | [ , ] | [, ] |
| 1 | Error | error |
| 2 | Info | info, confirm, prompt, receive |
| 3 | Input | input, data, characteristics, contingents, prices, special, change, enter, pay*, free, read |
| 4 | Help | help |
| 5 | Save | save, send, put, take, submit |
| 6 | Edit | edit, update |
| 7 | Add | add, new, compose, create |
| 8 | Ok | ok |
| 9 | Cancel | cancel |
| 10 | Process | process, encrypt, sign, server*, returnMoney |
| 11 | Calculate | calculate |
| 12 | Validate | validation, verify |
| 13 | Navigate | navigate, link, overview, continue, pause, finish, release |
| 14 | Delete | delete |
| 15 | Get | get, open, request |
| 16 | Load | load |
| 17 | Select | select |
| 18 | Print | print |
| 19 | Login | log*in, log, sing*in, access |
| 20 | View | view, trace, monitor |

node identifier, and the second column is the target node identifier for the edge. Other columns represent the features of the edge if it exists. The node mapping output file shows the mappings of nodes defined in ESG and the node identifier generated by the mxe parser application.

### C. MODEL TRAINING

#### 1) SEAL-ESG

SEAL [47] is a specialized framework for link prediction. With an innovative approach, we transform the link prediction problem into a sub-graph classification problem. We extract a surrounding sub-graph at an n-hop distance for each edge and create negative samples containing faulty connections. These generated sub-graphs and node feature matrix (which includes k features for each node) are fed to a GNN for classification. This way, node features and graph structure are used during learning.

SEAL implementation consists of reading graph data and node attributes from a file, loading them into a compressed sparse column matrix, and sampling positive and negative train/test links from a loaded matrix. If embedding learning is enabled, node2vec [54] is used to create node information. If the library runs on training mode (the default behavior), then SEAL extracts its n-hop (via hop argument) for each target link, encloses the subgraph, and creates its node information matrix. SEAL uses a DGCNN [40] classification model. SEAL transforms the link prediction problem into a graph classification problem, and each subgraph (positive and negative samples) generated by SEAL passes through DGCNN for the classification task.

In DGCNN architecture [40], the Sort Pooling layer is the key innovation, which differentiates it from other GCNs. On traditional GCN, feature values of neighboring nodes are summed up before passing them to CNN, but in DGCNN, the Sort Pooling layer organizes node features in a solid order. In this way, it makes it possible to keep more information about different node features. The input of this layer is node features and feature channels, and the output is sorted node features and output channels of each feature.

Original SEAL implementation is extended and customized as follows. Parameters of the DGCNN model are hidden and cannot be tuned externally. The ability to adjust the hyperparameters of a neural network is crucial and therefore it is added. Some minor bugs are fixed, which prevented the application from running on training data format except mat file format. The application used to printing training, validation, and test results on the screen. Working in this way was challenging to evaluate results between iterations. For this reason, all the results are now written in a CSV formatted file at the end of each iteration. The extended version of the SEAL is published to GitHub as SEAL-ESG [55] and can be accessed publicly. Available parameters of the SEAL-ESG implementation and their explanations are given at [56]. If embeddings are enabled, node2vec software is needed to run the application.

#### 2) DeepLinker-ESG

DeepLinker [48] is an extension of GAT [43], which specializes in link prediction. The input of a GAT is the features of each node, and the output is the learned features of each node produced by the GAT. A shared linear transformation with a weight matrix applied to each node is required to transform the input node features into a learned output feature. A single-layer feed-forward neural network (FFNN) with a weight vector called attention mechanism

**TABLE 3.** Node name to node feature mappings.

| Parameter Name | SEAL-ESG | DeepLinker-ESG |
|---|---|---|
| Batch size | X | X |
| Dropout Ratio | X | X |
| Number of Hidden Units | X | X |
| Learning Rate | X | X |
| Number of Epochs | X | X |
| Test Ratio | X | X |
| Hop | X | - |
| SortPooling K | X | - |
| Number of Attention | - | X |
| Weight Decay | - | X |

is used to determine which neighbors of a node are more important (softmax function is used for ranking).

DeepLinker creates a data set for a given graph by creating positive (nodes with connections) and negative (nodes with no connection) edge samples. The following operations are performed for each of the node pairs in the data set; for the current node pair (for example, 1 and 2), find the first (3, 4) and second level (1, 2, 5) neighbors of each node. DeepLinker uses fixed-sized neighborhood sampling for optimum memory usage and then calculates the edge vector representation of the node pair over their and their neighbor's initial features using GAT. After that, DeepLinker calculates the Hadamard distance of the GAT output, an edge vector representation of the node pair, and makes link predictions via training a logistic regression function.

Original DeepLinker implementation is extended and customized as follows. There was no parametric data input support to work with other training data. A feature that can load the outputs of the mxe parser application has been added. Only GPU support was available, and we added CPU support. Test evaluation metrics are calculated at the end of each epoch. The application prints training, validation, and test results on the screen. All the results are now written in a CSV formatted file at the end of each iteration. The extended version of the DeepLinker is published to GitHub as DeepLinker-ESG [57] and can be accessed publicly. Available parameters of the DeepLinker-ESG (also used for GAT) implementation are given at [58].

#### 3) SEAL-ESG AND DEEPLINKER-ESG PARAMETERS

Two different models are used in this step: SEAL-ESG and DeepLinker-ESG. To run on isolated environments, we created a Conda virtual environment (detailed information can be reached at [59]) for each workspace. Before using the virtual environments for SEAL-ESG, python version should be set to 3.8 and for the DeepLinker-ESG to 2.7. The tuned parameters of SEAL-ESG and DeepLinker-ESG used in experiments are given in Table 3.

## V. EVALUATION

SEAL-ESG and DeepLinker-ESG link prediction approaches are performed on ESG models; Bank Account, Student Attendance, Email, and ISELTA drawn by the TSD tool. The studies were conducted to predict possible missing links on a given ESG. In addition, results and discussion, threats to validity are explained in this section. The experiment steps can be listed as follows: preparing the environment, determining node features and creating an embedding file to find node features, parsing the files with mxe extension, transforming them into files containing the edge and node information of the graph, and training the model using these output files. The experiments' environment details, including hardware configuration used, installed software, Python libraries, and GitHub repositories, are listed at [60].

### A. RESULTS

Parameter value tables for SEAL-ESG and DeepLinker-ESG are given at [61] and [62], respectively. Each model's first five parameters (batch size, dropout, hidden units, learning rate, and the number of epochs) are the same, typical for many neural networks. The remaining parameters are hops and sortpooling K parameters for SEAL-ESG and the number of attentions and weight decay for DeepLinker-ESG. The values of parameters used in each iteration for the training of SEAL-ESG and DeepLinker-ESG models are listed at [61] and [62], respectively.

For SEAL-ESG, batch size iterates among the values of 1, 10, 25, 50 (40 for the E-Mail data set) while dropout is held at 0.5. Hidden units are 64 or 128. The learning rate iterates among the values of 0.001, 0.0005, and 0.0001, while the number of epochs is kept at 50. Hops are either 1 or 2, and sortpooling K is kept at 0.6.

For DeepLinker-ESG, batch size is either 16 or 32, while dropout is held at 0.5 and hidden units at 32. The learning rate iterates among the values of 0.001, 0.0005, and 0.0001, while the number of epochs is kept at 50. The number of attentions is 2 or 8, and the weight decay is 0.001 or 0.0001.

All four datasets' best-performed results of SEAL-ESG iterations are listed in Table 4—performance of the SEAL-ESG training for each iteration measured by loss, accuracy, and AUC. Table 4 also gives the best-performed iteration number, which is 5 for ISELTA, 2 for Student, 4 for Bank, and 4 for E-mail data sets. ISELTA's best performance is loss with 0.306, acc with 0.875, and AUC with 0.957, while for Student, loss is 0.467, acc is 0.840, and AUC is 0.862. Bank has best performance at 0.331 for loss, at 0.868 for acc, and at 0.932 for AUC while E-mail's best performance is loss with 0.378, acc with 0.900, and AUC with 0.920. Please note that NaN stands for ''Not a valid Number'', a numerical overflow or underflow often referred to as ''exploding gradients'' and occurs due to extensive weight updates during training [63]. The rest of the iteration results are at [64].

All four datasets' best-performed results of DeepLinker-ESG iterations are listed in Table 5—performance of the DeepLinker-ESG training for each iteration measured by loss, accuracy, and AUC (only available for testing). Table 5 also gives the best-performed iteration number, which is 8 for all four data sets. ISELTA's best performance is loss

**TABLE 4.** SEAL-ESG best performed iteration results.

| Dataset | Best Iteration | loss | acc | AUC | loss | acc | AUC | loss | acc | AUC |
|---|---|---|---|---|---|---|---|---|---|---|
| ISELTA | 5 | 0.360 | 0.856 | 0.921 | 0.306 | 0.875 | 0.957 | 0.462 | 0.800 | 0.884 |
| Student | 2 | 0.467 | 0.840 | 0.862 | 0.721 | 0.667 | 0.500 | 0.613 | 0.719 | 0.740 |
| Bank | 4 | 0.331 | 0.868 | 0.932 | 0.451 | 0.800 | NaN | 0.617 | 0.786 | 0.755 |
| Email | 4 | 0.085 | 0.977 | NaN | 0.085 | NaN | NaN | 0.378 | 0.900 | 0.920 |

**TABLE 5.** DeepLinker-ESG best performed iteration results.

| Dataset | Best Iteration | loss | acc | AUC | loss | acc | AUC | loss | acc | AUC |
|---|---|---|---|---|---|---|---|---|---|---|
| ISELTA | 8 | 0.693 | 0.552 | NaN | 0.652 | 0.875 | NaN | 0.688 | 0.594 | 0.587 |
| Student | 8 | 0.684 | 0.574 | 0.000 | 0.646 | 0.571 | 0.000 | 0.683 | 0.625 | 0.625 |
| Bank | 8 | 0.675 | 0.643 | NaN | 0.671 | 0.667 | NaN | 0.681 | 0.781 | 0.703 |
| Email | 8 | 0.676 | 0.594 | 0.000 | 0.668 | 1.000 | 0.000 | 0.687 | 0.607 | 0.577 |

with 0.688, acc with 0.594, and AUC with 0.587, while for Student, loss is 0.683, acc is 0.625, and AUC is 0.625. Bank has best performance at 0.681 for loss, at 0.781 for acc, and at 0.703 for AUC while E-mail's best performance is loss with 0.687, acc with 0.607, and AUC with 0.577. The rest of the iteration results are at [65].

SEAL-ESG outperforms DeepLinker-ESG in all datasets.

### B. DISCUSSION

The experiments are performed on the four datasets explained above. Each of these ESG models has its specific domain, and the components of these software models are observed to contain *particular patterns*. It is considered that these patterns can be revealed through graph neural networks, which are specialized for graph-structured data. The experiments are performed under these considerations.

We explain *particular patterns* with an example. For instance, the simplified Bank Account ESG in Figure 2 has a pattern. After the ''get balance'' event, the user can select an operation and needs to enter an amount related to that operation. The upper half of the ESG has the ''select deposit''-''enter deposit amount'' event sequence, and the bottom half of the ESG has the ''select withdraw''-''enter withdraw amount'' event sequence. There is a pattern of event sequence such as ''select [operation]''-''enter [operation] amount'' where [operation] can be considered as a variable. This pattern is an observable pattern particular to the bank account domain. Imagine that the link between ''select withdraw'' and ''enter withdraw amount'' events (nodes) is missing. For a human, it is immediately recognizable that the link is missing. With this research, we aim to mimic this human recognition with graph neural networks. When an ESG is small, as in Figure 2, it is easy for humans to recognize missing links. However, it can be challenging for humans to recognize when the ESG is as big as ISELTA ESG, with 68 nodes and 249 edges (links). In this case, our approach provides a solution.

First impressions of the experimental results are as follows: SEAL-ESG uses DGCNN as a GNN model under the hood. It converts the link existence problem into a sub-graph classification problem by dividing a given graph into sub-graphs (with samples created with negative and positive neighbors for each node). It performed much better than DeepLinker-ESG (which uses GAT as a GNN model), trying to solve the link existence problem by learning the hidden representations of nodes' relations with their neighbors.

Before evaluating the experiments' results, it is necessary to briefly mention how the metrics are used in assessing the results. The area under curve (AUC) can be considered the summary of the model performance and gives classes within the dataset for all classification thresholds. The wider the area under the roc (receiver operating characteristic) curve, the higher the model's ability to distinguish classes. An AUC value of 0.5 means random estimation. The closer this value is to 1, the higher the model's ability to differentiate between classes. Acc (accuracy) is the primary performance metric that expresses the number of observations made correctly with respect to the model. Still, in most cases, it is not sufficient to measure the model's performance alone (for example, where the distribution of the dataset between classes is not balanced). Loss (cross-entropy) gives the difference between the estimation made by the model and the actual value. Classification results generated by a neural network fall into [0,1] interval for each class. The neural network model assigns a value between [0,1] for each class based on the input values. The class with the highest value is taken as the result of these assigned values. While the accuracy metric evaluates the results as true or false, the loss metric measures how far the model's value for the correct class is from 1.

The performance outputs of the parameters used in SEAL-ESG iterations are given in Figure 4. Regardless of the datasets' size, iterations 9, 21, and 29 show the worst performance. As the dataset gets smaller, the performance was negatively affected in all iterations between 17 and 32. Looking at the effects of the hop parameter on performance, we can say that for all the first-order neighbors of a component belonging to a software model, the representation is learned best by DGCNN. A significant case occurs in the 16th iteration, setting the batch size to the minimum value of 1 and the learning rate to 0.001 (the most significant learning rate used in experiments). Even though the model overfits large datasets, a small positive effect is enhanced on performance in small datasets. When the iterations with the best results are examined, the performance is higher in iterations 1, 2, 5, 8, 11, and 26 in large datasets, while

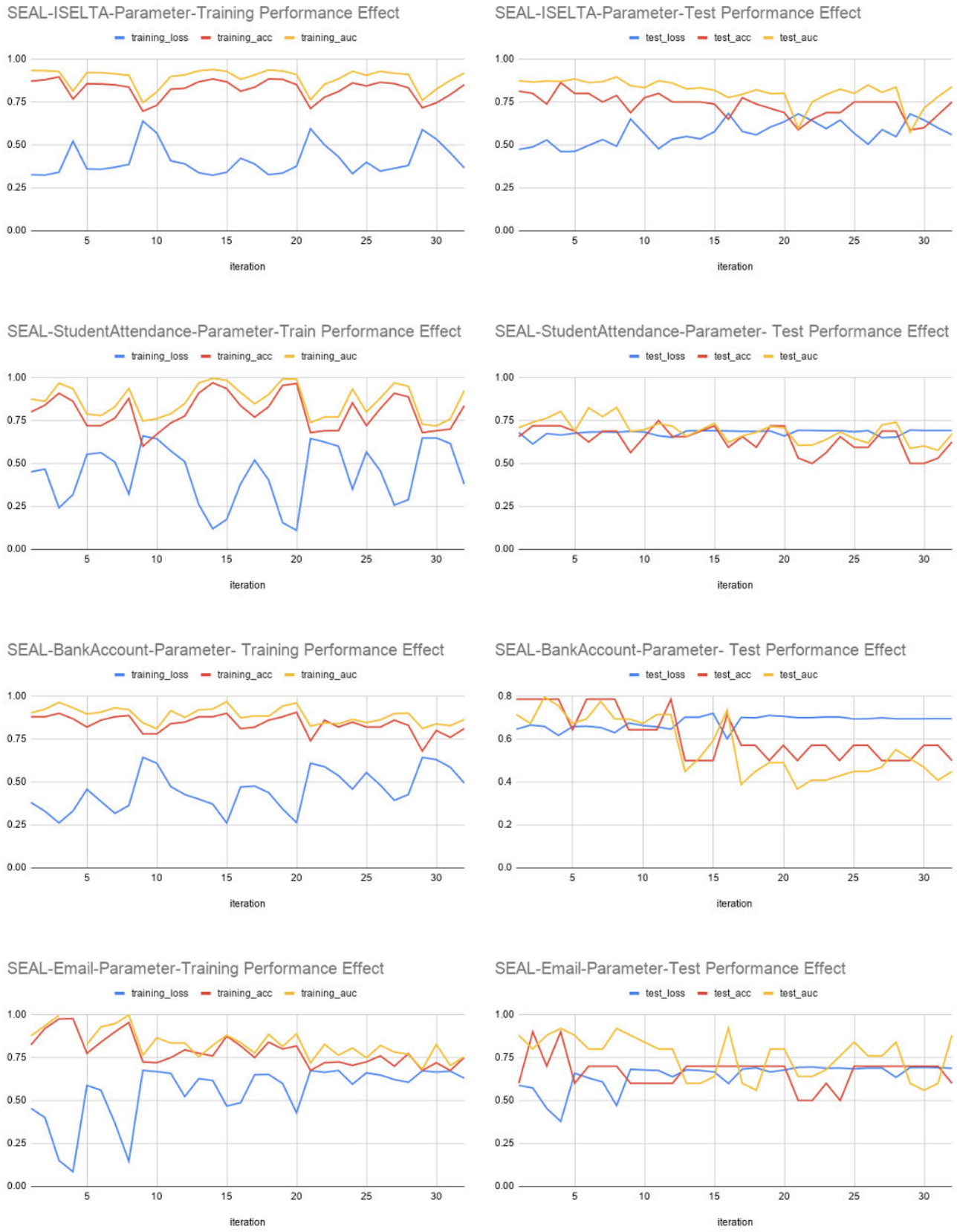


**FIGURE 4.** DeepLinker-ESG performance effects of parameter changes on each iteration.

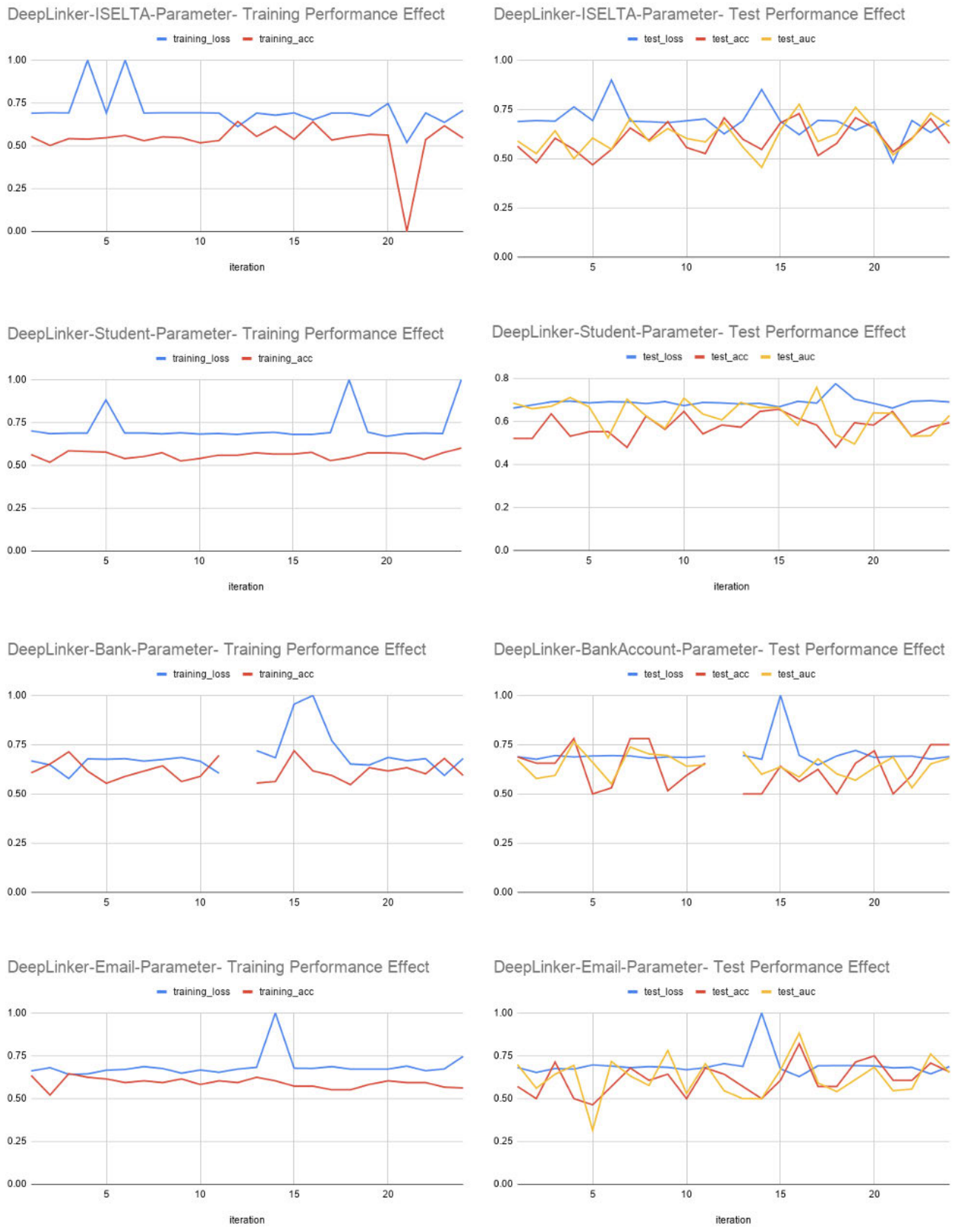


**FIGURE 5.** SEAL-ESG performance effects of parameter changes on each iteration.

iterations 3, 4, 7, 12, and 16 showed higher performance in smaller iterations. Examining the findings reveals that setting the batch size to 1 increases the probability of overfitting. It is observed that changing the batch size value and the learning rate values inversely increases the performance. As the dataset grows, using a larger value batch size and a lower learning rate positively affects the performance.

When the parameters used in DeepLinker-ESG iterations and the performance outputs of these parameters are compared in Figure 5, performance is distributed around 0.5, close to random estimation, even if the tunings are performed by changing the parameters, it makes $+-10\%$ performance changes. While the model was being trained, the distribution of the dataset between negative and positive classes (negative meaning no link and positive meaning there is a link between nodes) was made equally. At the same time, we adjusted the distribution within batches to be equal. As a result, it can be thought that software models are relatively small models, and there is not enough data for GAT to learn the relationships between nodes. When the datasets used in the article where the GAT model is used for link prediction are examined, it is seen that large-scale graphs are used. For example, the CorA dataset in the article [28] consists of 2708 nodes, 5429 edges, and 1433 node features. On the other hand, ISELTA, the largest dataset used in this work, has 68 nodes, 249 edges, and one node feature.

Experiments on two different machine learning models with four datasets have shown that one of the best ways to understand how nodes are used in graphical software models is to form a pattern with their neighboring nodes through the sub-graphs (i.e., micro-models). This way, we obtained successful results even with relatively small datasets.

**SEAL-ESG results in better performance predicting missing links between ESG nodes than DeepLinker-ESG.**

The disadvantage of SEAL-ESG is that when a disconnected graph is given, it is impossible to make an edge prediction from scratch (without any edge definition) since it cannot generate sub-graphs for this graph.

### C. EXAMPLES OF LINK PREDICTION

The link prediction examples for ISELTA-Specials ESG given in Figure 6 are shown in Figure 7 and Figure 8. A SEAL-ESG model is trained using the ISELTA dataset. This trained model executes link prediction scenarios for the nodes ''edit Special'' and ''delete Special.'' Green dotted arrows are the new links predicted by the trained model that is not defined in the original ESG.

For the ''edit Special'' node, link predictions and the probabilities generated by the trained model are listed in Table 6, given in Figure 7.

**TABLE 6. Link predictions made by the trained model for "edit special" node.**

| Link | Probability Of Existence |
|---|---|
| '[', 'edit Special' | %79 |
| 'Ok', 'edit Special' | %73 |
| 'add', 'edit Special' | %60 |
| 'edit Special', 'save' | %73 |
| 'edit Special', 'SpecialData2' | %78 |
| 'edit Special', 'cancel' | %49 |
| 'cancel', 'edit Special' | %73 |

**TABLE 7. Link predictions made by the trained model for "delete special" node.**

| Link | Probability Of Existence |
|---|---|
| '[', 'delete Special' | %95 |
| 'delete Special', 'Ok' | %94 |
| 'delete Special', 'Cancel' | %78 |
| 'save', 'delete Special' | %78 |
| 'SpecialData2', 'delete Special' | %70 |
| 'add', 'delete Special' | %61 |
| 'cancel', 'delete Special' | %31 |

For the ''delete Special'' node, link predictions and the probabilities generated by the trained model are also listed in Table 7 shown in Figure 8.

The results suggested by the model can be evaluated as follows. There are two new possible connection suggestions for the ''edit Special'' node with probabilities of 79% and 73%. These suggestions should be taken into consideration by the modeler. Besides, a suggestion with a probability value of 49% is presented for the connection between ''edit Special'' and ''cancel'' nodes. This suggestion may be thought of as ''do not care,'' The connection can be left as it is or removed at the modeler's discretion. For the 'delete Special' node, it is seen that two new connections and one low-probability connection are offered. The connection from ''cancel'' to ''delete Special'' has a probability value of 31%. It may be considered to break this existing connection entirely.

Prediction of missing links in graphical software models is a part of identifying missing requirements/specifications. Industry-standard practices to identify and address these missing specifications include structured requirements elicitation with stakeholders, analysis of use cases or user stories through reviews and inspections, manual checking models/diagrams, and prototyping. These practices are not automated and are performed manually. Our approach automatically predicts missing links and shows them with a percentage. It means that the trained GNN model suggests a possibility that that link is missing. The owner of ESG, either a person or a team, may accept or reject the suggestion. When we checked the results of our prediction approach, the ones with high percentages made sense, and it is worthwhile to consider them.

### D. IMPLICATIONS

Predicting missing links in graphical software models can have a range of implications. Since the fundamental objective

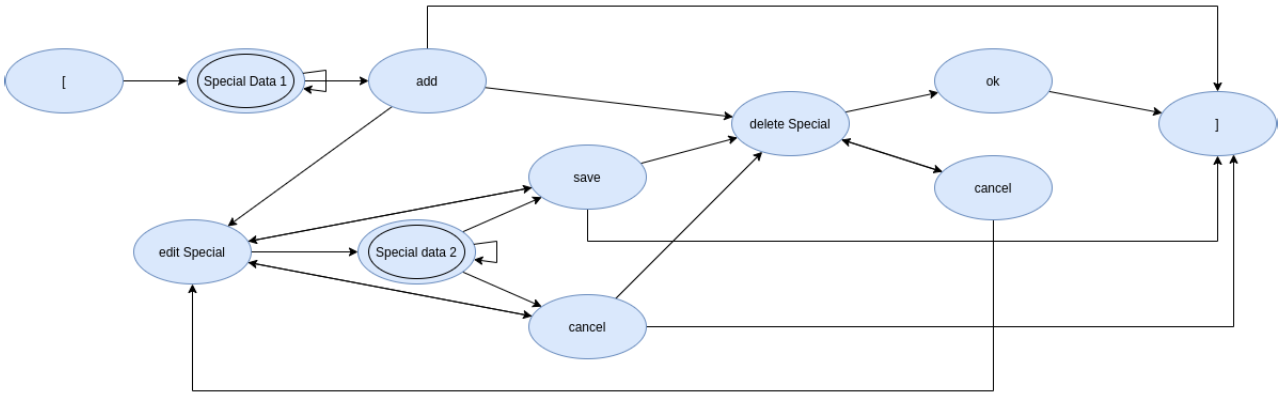


**FIGURE 6.** Original ISELTA-Specials ESG.

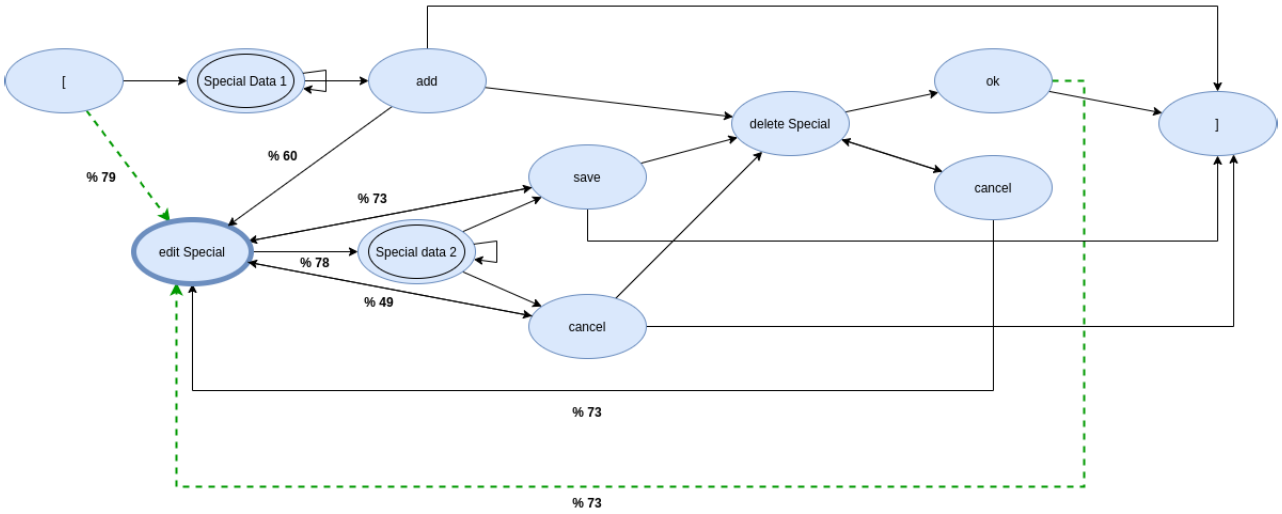


**FIGURE 7.** ISELTA-Specials ESG "edit special" node qualitative link predictions.

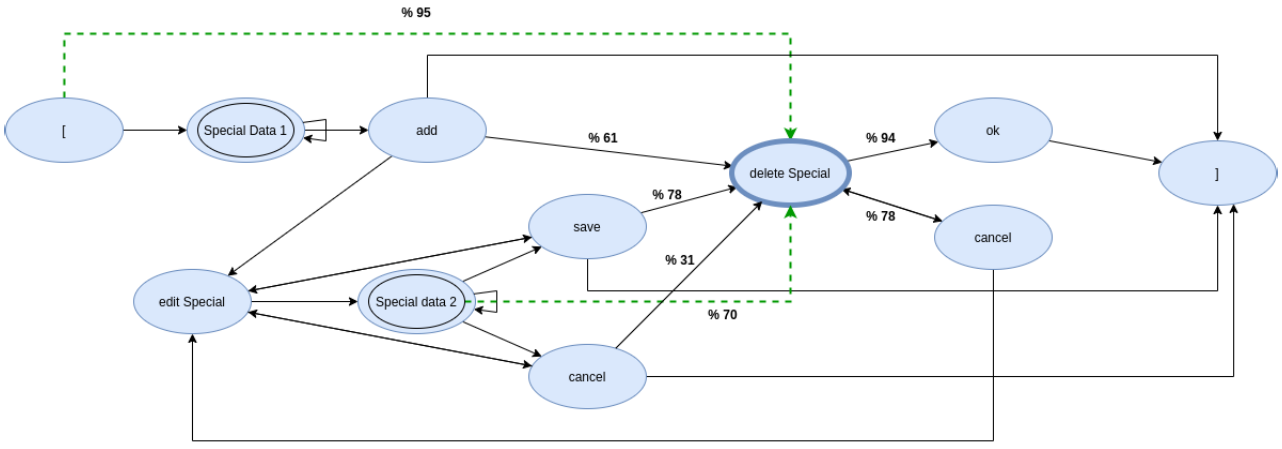


**FIGURE 8.** ISELTA-Specials ESG "delete special" node qualitative link predictions.

of a software model is to represent some aspect of the software, in the event of missing links, the model's capacity to offer a comprehensive and precise representation is compromised, potentially resulting in misunderstandings or misguided decisions. Those misunderstandings or misguided decisions can result in software development with missing features or functionalities. Conversely, software engineers might also build redundant or unnecessary features or

functionalities. Moreover, missing links might lead to integrity and consistency issues, compromising the quality of the software.

Both the validation and verification processes are dependent on having accurate models. The absence of some links can make these processes more difficult and lead to inaccurate or insufficient testing. Since test generation is automatic from ESG models, missing links will change test sequences, which will have a direct impact on the quality of the software. Furthermore, an incomplete model might cause complications when it comes to maintenance. Without a comprehensive understanding of the software, maintenance tasks might become more laborious and prone to error.

Errors or omissions caused by missing links can result in increased costs in terms of time and resources, as errors may need to be resolved later in the development cycle. Decisions, resource allocations, and approvals all rely on reliable models. Stakeholders can be misled into making poor decisions if critical links are missing. In cases where contracts or legal agreements regulate software development, an incomplete model could breach those requirements and result in legal ramifications.

In conclusion, missing links in graphical software models can cause many problems, from technical issues to financial costs. It is crucial to ensure that models are as complete and accurate as possible, are reviewed often, and are updated when the system changes.

### E. THREATS TO VALIDITY

Threats to validity are summarized under construct validity, internal validity and external validity.

**Construct Validity:** Since the number datasets is four, experiment results may not reasonably represent that missing link prediction ability of SEAL-ESG and DeepLinker-EESG.

**Internal Validity:** Datasets are selected from different domains and different sizes of software applications to make the evaluations trustworthy. Two GNN models, which apply to link prediction problems, are selected for experiments. The performance of these models is measured with a different set of parameter values. All the software applications are modeled by ESG and drawn by TSD.

**External Validity:** It is unlikely to say that the proposed method will work on different software modeling tools and processes, even if it is possible. Considering a class diagram modeled with UML notation, they are heterogenous directed multigraphs, but ESGs are homogenous directed graphs. The connection between classes has completely different meanings in comparison with ESGs.

## VI. RELATED WORK

The primary objective of this work is to identify and locate absent components within graphical software models. One potential strategy is the utilization of machine learning techniques for link prediction in order to enhance the completeness of graphical software models. Another strategy involves the application of methods such as model mining and model checking to infer missing components within these models. The literature pertaining to each direction are delineated in the subsequent sections. The suggested methodology is evaluated against existing methodologies in completing graphical software models.

### A. MACHINE/DEEP LEARNING APPROACHES IN COMPLETING GRAPHICAL SOFTWARE MODELS

The link prediction problem is a long-standing challenge in modern information science, and algorithms based on Markov chains, random walk processes, maximum likelihood methods, and statistical models have been proposed [66]. Lately, machine learning and deep learning approaches [45], [46], [47], [48] have taken their place. Although machine learning and deep learning approaches have recently found applications in link prediction, mainly in social networks [67], [68], to the best of the authors' knowledge, this work is the first application of link prediction to graphical software models.

Social networks are very large graphical models [67] compared to graphical software models, which are small in nature. For social networks, scalable methods or sampling techniques are essential [68], whereas for graphical software models, such methods and techniques are not necessary. The evolution of link prediction via machine/deep learning explained in Section III, two state-of-the-art techniques, namely SEAL [47] and DeepLinker [48], that are suitable for small graphical models, are applied to ESGs in this study.

SEAL uses sub-graphs, attributes, and embedding features of the graph for link prediction. SEAL extracts sub-graphs of related nodes, learns the features of these sub-graphs via DGCNN [40] and uses the learned model for link prediction. DeepLinker, on the other hand, uses fixed neighborhoods on a mini-batch sampling strategy. DeepLinker shares a similar architecture with GraphSAGE [41] with differences in sampling strategy and using GAT [43] instead of GCN [69]. The DeepLinker model creates a hidden representation of each node using the attention mechanism shared by the node's neighbors.

SEAL-ESG results in better performance predicting missing links between ESG nodes than DeepLinker-ESG due to the observations made in experiments that one of the best ways to understand how nodes are used in graphical software models is to form a pattern with their neighboring nodes through the sub-graphs, i.e., micro-models. This way, we obtained successful results even with relatively small graphical models of software contrary to large graphical models of social networks.

### B. MODEL-BASED APPROACHES IN COMPLETING GRAPHICAL SOFTWARE MODELS

One research direction in this focus is business process model discovery. Rozinat and van der Aalst [70] used event logs to conform to the process model. They proposed two dimensions of conformance, namely fitness and appropriateness,

to be checked by implementing a conformance checker within the ProM Framework. Beschastnikh et al. [71] proposed algorithms for inferring communicating finite state machine models from traces of concurrent systems and algorithms to prove them correct. Pecchia et al. [72] proposed an approach that employs process mining to discover process models from logs; then, it uses conformance checking to detect deviations from the discovered models. They were able to quantify the failure detection capability of conformance checking despite missing events and its accuracy for the process models obtained from noisy logs [72].

Another research direction in predicting missing model parts is model checking to find missing properties of software design models. Schäfer et al. [73] proposed an approach to verify whether a set of UML state machines can realize the interactions expressed by a UML collaboration. For this purpose, they compiled state machines into a PROMELA model and collaborations into sets of Büchi automata. They utilized the SPIN model checker to verify the model against the automata. Bentahar et al. [74] proposed a model checking-based approach for composite web services where the operational behavior is the model to be checked against properties defined in the control behavior. The operational behavior defines the composition functioning according to the Web services' business logic, and a control behavior identifies the valid sequences of actions that the operational behavior should follow [74]. These two behaviors are formally defined using automata-based techniques and then model-checked for missing properties [74].

In these two research directions, proposed solutions utilize a different second model or a different software artifact to find missing model parts. In our approach, we do not use a different second model or a different software artifact but instead utilize the same model to explore patterns and predict missing model parts.

## VII. CONCLUSION

Enterprise software applications are generally sophisticated. Such systems can have many sub-systems and components in them. The details of the software must be understood at varying levels of specification and design. Typical software modeling systems may not be able to reduce this complexity for engineers. Predicting the connections between software components has great importance in modeling. In software engineering, deciding on and connecting components requires significant effort. It is also error-prone. Instead of putting all the workload on software engineers' shoulders, giving some recommendations can help engineers model the composition and interaction of events, objects, and components in a software system.

The proposed method aims to help software engineers with software modeling. This work's modeling technique is ESG, which models the transition between GUI components. This paper presents a method to find missing links between components defined in ESG. Graph neural network models are used to solve this problem. Selected GNN models are graph convolutional neural networks and graph attention neural networks. To find missing links between nodes of an ESG model, we first transformed ESG models into graph-structured data and extracted features of the nodes. Then, we trained the GNN model and evaluated the performance of the trained model. Through the evaluation, we found the best hyperparameters for the best performance.

Experiments are performed on four datasets with two different GNN models, namely SEAL-ESG and DeepLink-ESG. The results show that it is possible to make recommendations on missing links or edges of the graph-based system model. SEAL-ESG results in better performance predicting missing links between ESG nodes than DeepLink-ESG.

This research is focused on ESG models to find missing links between components. Four datasets are used in this study. Diversifying datasets and evaluating their results in larger datasets could be subject of future studies. There are many software modeling tools and methods in the literature. We plan to work on other methodologies used for software modeling in the future. As another application area, our work can increase the accuracy of models created automatically with the ripping method. Another future work is to be able to make missing link predictions with disconnected ESGs. These predictions may be projected with the GNN model, which learns the feature representation of these nodes and performs link prediction by learning the relationships between the hidden representation of the nodes, not the edges.

## REFERENCES

[1] S. Robertson and J. Robertson, *Mastering the Requirements Process: Getting Requirements Right*. Reading, MA, USA: Addison-Wesley, 2012.
[2] J. Krogstie, G. Sindre, and H. Jørgensen, "Process models representing knowledge for action: A revised quality framework," *Eur. J. Inf. Syst.*, vol. 15, pp. 91–102, Feb. 2006.
[3] T. Stahl, M. Völter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ, USA: Wiley, 2006.
[4] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ, USA: Wiley, 2013.
[5] L. Aversano, C. Grasso, and M. Tortorella, "Managing the alignment between business processes and software systems," *Inf. Softw. Technol.*, vol. 72, pp. 171–188, Apr. 2016.
[6] F. Belli, M. Beyazit, and N. Güler, "Event-based GUI testing and reliability assessment techniques—An experimental insight and preliminary results," in *Proc. IEEE 4th Int. Conf. Softw. Test., Verification Validation Workshops*, Mar. 2011, pp. 212–221.
[7] M. Stephan, "Towards a cognizant virtual software modeling assistant using model clones," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., New Ideas Emerg. Results (ICSE-NIER)*, May 2019, pp. 21–24.
[8] G. Mussbacher, B. Combemale, J. Kienzle, S. Abrahão, H. Ali, N. Bencomo, L. Burgueño, G. Engels, and P. Jeanjean, "Opportunities in intelligent modeling assistance," *Softw. Syst. Model.*, vol. 19, no. 5, pp. 1045–1053, Sep. 2020.
[9] A. Forward, O. Badreddin, and T. C. Lethbridge, "Perceptions of software modeling: A survey of software practitioners," in *Proc. From Code Centric Model Centric, Evaluating Effectiveness MDD (C2M:EEMDD)*, Paris, France, 2010.
[10] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio, "Relevance, benefits, and problems of software modelling and model driven techniques—A survey in the Italian industry," *J. Syst. Softw.*, vol. 86, no. 8, pp. 2110–2126, Aug. 2013.

[11] G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. 1. Singapore: World Scientific, 1997.

[12] J.-L. Pérez-Medina, S. Dupuy-Chessa, and A. Front, "A survey of model driven engineering tools for user interface design," in *Proc. Int. Workshop Task Models Diagrams User Interface Design*. Toulouse, France: Springer, 2007, pp. 84–97.

[13] R. F. Paige, N. Matragkas, and L. M. Rose, "Evolving models in model-driven engineering: State-of-the-art and future challenges," *J. Syst. Softw.*, vol. 111, pp. 272–280, Jan. 2016.

[14] G. Taentzer, F. Mantz, T. Arendt, and Y. Lamo, "Customizable model migration schemes for meta-model evolutions with multiplicity changes," in *Model-Driven Engineering Languages and Systems*. Miami, FL, USA: Springer, 2013, pp. 254–270.

[15] C. Krause, J. Dyck, and H. Giese, "Metamodel-specific coupled evolution based on dynamically typed graph transformations," in *Proc. Int. Conf. Theory Pract. Model Transformations*. Cham, Switzerland: Springer, 2013, pp. 76–91.

[16] A. Memon, I. Banerjee, B. N. Nguyen, and B. Robbins, "The first decade of GUI ripping: Extensions, applications, and broader impacts," in *Proc. 20th Work. Conf. Reverse Eng. (WCRE)*, Oct. 2013, pp. 11–20.

[17] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sep. 2012, pp. 258–261.

[18] D. Lo and S.-C. Khoo, "Software specification discovery: A new data mining approach," Dept. Comput. Sci., Nat. Univ. Singapore, NSF NGDM, Tech. Rep., 2007.

[19] T. Kanstrén, É. Piel, and H.-G. Gross, "Observation-based modeling for model-based testing," Softw. Eng. Res. Group, Delft Univ. Technol., Tech. Rep. TUD-SERG-2009-012, 2009.

[20] M. G. Gabel, *Inferring Programmer Intent and Related Errors From Software*. Davis, CA, USA: Univ. California, Davis, 2011.

[21] M. T. Ailane, A. Aniculaesei, C. Knieke, A. Rausch, and F. Sholichin, "Towards specification completion for systems with emergent behavior based on DevOps," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2022, pp. 1830–1836.

[22] R. H. Thayer, S. C. Bailin, and M. Dorfman, *Software Requirements Engineerings*. Washington, DC, USA: IEEE Computer Society Press, 1997.

[23] U. S. Shah and D. C. Jinwala, "Resolving ambiguities in natural language software requirements: A comprehensive survey," *ACM SIGSOFT Softw. Eng. Notes*, vol. 40, no. 5, pp. 1–7, Sep. 2015.

[24] Q. Zhi, L. Gong, J. Ren, M. Liu, Z. Zhou, and S. Yamamoto, "Element quality indicator: A quality assessment and defect detection method for software requirement specification," *Heliyon*, vol. 9, no. 5, May 2023, Art. no. e16469.

[25] O. Y. Leblebici, "Application of graph neural networks on software modeling," Ph.D. dissertation, Izmir Inst. Technol. (Turkey), Urla, Turkiye, 2020.

[26] A. Khalilov, T. Tuglular, and F. Belli, "Mutation analysis of specification-based contracts in software testing," in *Proc. 15th Turkish Nat. Softw. Eng. Symp. (UYMS)*, Nov. 2021, pp. 1–6.

[27] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, and W. E. Wong, "Model-based mutation testing—Approach and case studies," *Sci. Comput. Program.*, vol. 120, pp. 25–48, May 2016.

[28] F. Belli, "Finite state testing and analysis of graphical user interfaces," in *Proc. 12th Int. Symp. Softw. Rel. Eng.*, 2001, pp. 34–43.

[29] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: An innovative tool for automated testing of GUI-driven software," *Automated Softw. Eng.*, vol. 21, no. 1, pp. 65–105, Mar. 2014.

[30] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Hierarchical GUI test case generation using automated planning," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 144–155, Feb. 2001.

[31] F. Belli, M. Beyazit, A. T. Endo, A. Mathur, and A. Simao, "Fault domain-based testing in imperfect situations: A heuristic approach and case studies," *Softw. Quality J.*, vol. 23, pp. 423–452, Sep. 2015.

[32] F. Belli and A. Hollmann, "Test generation and minimization with 'basic' statecharts," in *Proc. ACM Symp. Appl. Comput.*, 2008, pp. 718–723.

[33] J. Myhill, "Finite automata and the representation of events," *WADD Tech. Rep.*, vol. 57, pp. 112–137, Jan. 1957.

[34] F. Belli and C. J. Budnik, "Test minimization for human-computer interaction," *Int. J. Speech Technol.*, vol. 26, no. 2, pp. 161–174, Mar. 2007.

[35] F. Belli, C. J. Budnik, and L. White, "Event-based modelling, analysis and testing of user interactions: Approach and case study," *Softw. Test., Verification Rel.*, vol. 16, no. 1, pp. 3–32, 2006.

[36] D. Ozturk, "A model-based test generation approach for agile software product lines," Ph.D. dissertation, Izmir Inst. Technol. (Turkey), Urla, Turkiye, 2020.

[37] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Dec. 2009.

[38] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," in *The Handbook of Brain Theory and Neural Networks*, vol. 3361. Cambridge, MA, USA: MIT Press, 1995.

[39] D. K. Hammond, P. Vandergheynst, and R. Gribonval, "Wavelets on graphs via spectral graph theory," *Appl. Comput. Harmon. Anal.*, vol. 30, no. 2, pp. 129–150, Mar. 2011.

[40] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Proc. AAAI Conf. Artif. Intell.*, 2018, vol. 32, no. 1, pp. 1–8.

[41] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.

[42] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," 2018, *arXiv:1801.10247*.

[43] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," 2017, *arXiv:1710.10903*.

[44] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D.-Y. Yeung, "GaAN: Gated attention networks for learning on large and spatiotemporal graphs," 2018, *arXiv:1803.07294*.

[45] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *J. Amer. Soc. Inf. Sci. Technol.*, vol. 58, no. 7, pp. 1019–1031, 2007.

[46] I. A. Kovács, K. Luck, K. Spirohn, Y. Wang, C. Pollis, S. Schlabach, W. Bian, D.-K. Kim, N. Kishore, and T. Hao, "Network-based prediction of protein interactions," *Nature Commun.*, vol. 10, no. 1, pp. 1–8, 2019.

[47] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, 2018, pp. 1–11.

[48] W. Gu, F. Gao, X. Lou, and J. Zhang, "Link prediction via graph attention network," 2019, *arXiv:1910.04807*.

[49] R. Jacobs, T. Mayeshiba, B. Afflerbach, L. Miles, M. Williams, M. Turner, R. Finkel, and D. Morgan, "The materials simulation toolkit for machine learning (MAST-ML): An automated open source toolkit to accelerate data-driven materials research," *Comput. Mater. Sci.*, vol. 176, Apr. 2020, Art. no. 109544.

[50] T. Shilpa and A. Bai, "A review on cardiovascular disease detection using machine learning algorithms," *Solid State Technol.*, vol. 63, no. 5, pp. 8754–8768, 2020.

[51] T. Tuglular, F. Belli, and M. Linschulte, "Input contract testing of graphical user interfaces," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 26, no. 2, pp. 183–215, Mar. 2016.

[52] Y. Bengio, R. Ducharme, and P. Vincent, "A neural probabilistic language model," *Adv. Neural Inf. Process. Syst.*, vol. 13, 2000, pp. 1–7.

[53] O. Leblebici. *Python Listing Flatten Graph in Application of GNN on Software Modeling*. Accessed: Apr. 20, 2023. [Online]. Available: https://github.com/onurleblebici/ApplicationOfGNNOnSoftware Modeling/blob/master/pythonlisting_flattenGraph.pdf

[54] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 855–864.

[55] O. Leblebici. *Seal-ESG—An Implementation of Seal for Event Sequence Graph Link Prediction Tasks*. Accessed: Apr. 20, 2023. [Online]. Available: https://github.com/onurleblebici/SEAL-ESG

[56] O. Leblebici. *Arguments of the Seal-ESG Framework*. Accessed: Apr. 20, 2023. [Online]. Available: https://github.com/onurleblebici/ApplicationOfGNNOnSoftwareModeling/blob/master/AppendixA.pdf

[57] O. Leblebici. *Deeplinker-ESG—An Implementation of Deeplinker for Event Sequence Graph Link Prediction Tasks*. Accessed: Apr. 20, 2023. [Online]. Available: https://github.com/onurleblebici/DeepLinker-ESG

[58] O. Leblebici. *Parameters of the Deeplinker*. Accessed: Apr. 20, 2023. [Online]. Available: https://github.com/onurleblebici/ApplicationOfGNNOnSoftwareModeling/blob/master/AppendixB.pdf

[59] *Conda Environments*. Accessed: Apr. 20, 2023. [Online]. Available: https://docs.conda.io/projects/conda/en/latest/user-guide/concepts/environments.html

[60] O. Leblebici. *Computer Hardware Specifications*. Accessed: Apr. 20, 2023. [Online]. Available: https://github.com/onurleblebici/ApplicationOfGNNOnSoftwareModeling/blob/master/AppendixC.pdf

[61] O. Leblebici. *Seal-ESG Parameters on Each Iteration*. Accessed: Apr. 20, 2023. [Online]. Available: https://github.com/onurleblebici/ApplicationOfGNNOnSoftwareModeling/blob/master/AppendixD.pdf
[62] O. Leblebici. *Deeplinker-ESG Parameters on Each Iteration*. Accessed: Apr. 20, 2023. [Online]. Available: Available: https://github.com/onurleblebici/ApplicationOfGNNOnSoftware Modeling/blob/master/AppendixD.pdf
[63] J. Brownlee. *How to Avoid Exploding Gradients With Gradient Clipping*. Accessed: Apr. 20, 2023. [Online]. Available: https://docs.conda.io/projects/conda/en/latest/user-guide/concepts/environments.html
[64] O. Leblebici. *Seal-ESG Dataset Results*. Accessed: Apr. 20, 2023. [Online]. Available: https://github.com/onurleblebici/SEAL-ESG/blob/main/iteration-result-sealESG.pdf
[65] O. Leblebici. *Deeplinker-ESG Dataset Results*. Accessed: Apr. 20, 2023. [Online]. Available: https://github.com/onurleblebici/DeepLinker-ESG/blob/main/iteration-result-deepplinkerESG.pdf
[66] L. Lü and T. Zhou, "Link prediction in complex networks: A survey," *Phys. A, Stat. Mech. Appl.*, vol. 390, no. 6, pp. 1150–1170, 2011.
[67] N. N. Daud, S. H. Ab Hamid, M. Saadoon, F. Sahran, and N. B. Anuar, "Applications of link prediction in social networks: A review," *J. Netw. Comput. Appl.*, vol. 166, Sep. 2020, Art. no. 102716.
[68] H. Yuliansyah, Z. A. Othman, and A. A. Bakar, "Taxonomy of link prediction for social network analysis: A review," *IEEE Access*, vol. 8, pp. 183470–183487, 2020.
[69] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.
[70] A. Rozinat and W. M. Van der Aalst, "Conformance testing: Measuring the fit and appropriateness of event logs and process models," in *Proc. Int. Conf. Bus. Process. Manag.* Cham, Switzerland: Springer, 2005, pp. 163–176.
[71] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with CSight," in *Proc. 36th Int. Conf. Softw. Eng.*, May 2014, pp. 468–479.
[72] A. Pecchia, I. Weber, M. Cinque, and Y. Ma, "Discovering process models for the analysis of application failures under uncertainty of event logs," *Knowl.-Based Syst.*, vol. 189, Feb. 2020, Art. no. 105054.
[73] T. Schäfer, A. Knapp, and S. Merz, "Model checking UML state machines and collaborations," *Electron. Notes Theor. Comput. Sci.*, vol. 55, no. 3, pp. 357–369, Oct. 2001.
[74] J. Bentahar, H. Yahyaoui, M. Kova, and Z. Maamar, "Symbolic model checking composite web services using operational and control behaviors," *Exp. Syst. Appl.*, vol. 40, no. 2, pp. 508–522, Feb. 2013.

**ONUR LEBLEBICI** received the B.Sc. degree in computer engineering from Hacettepe University, in 2006, and the M.Sc. degree in computer engineering from the İzmir Institute of Technology, Turkey, in 2020. Since 2006, he has been a software engineer, a research and development manager, a software development manager, and a software development consultant in various national and international companies. He is currently the Chief Software Architect in Univera Company and responsible for managing research and development projects, designing and building brand new software development platforms and determining architectural compliance within the context of the system architecture and best practices. His research interests include enterprise business applications, software architectures, and model-driven engineering.

**TUGKAN TUGLULAR** (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer engineering from Ege University, Turkey, in 1993, 1995, and 1999, respectively. He was a Research Associate with Purdue University, from 1996 to 1998. He has been with the İzmir Institute of Technology, since 2000. After, he became an Assistant Professor with the İzmir Institute of Technology, where he was the Chief Information Officer, from 2003 to 2007. Currently, in addition to his academic duties, he acts as an IT Advisor to the Rector. He has more than 60 publications and active record of duties with international and national conferences. His current research interests include model-based testing and model-based software development.

**FEVZI BELLI** (Member, IEEE) received the B.S., M.S., Ph.D., and Habilitation (a German postdoctoral) degrees in information technology and computer science from Technical University Berlin. He is currently a Professor Emeritus in software engineering with the University of Paderborn and the İzmir Institute of Technology. He has more than 35 year's experience in research, development and teaching software engineering, validation and verification, fault tolerance, and quality assurance. He started as a programmer in the aircraft industry and wrote programs to create simulation environments and to validate safety critical features. In 1983, he was awarded a Professorship with the University of Applied Sciences in Bremerhaven, in 1989, he moved to the University of Paderborn. He was also, for many years, a Faculty Member of the University of Maryland, College Park, MD, USA, and the European Division. He was also the Founding Chair of the Computer Science Department, University of Economics, İzmir, Turkey. His research interests include software reliability/fault tolerance, model-based testing, and test automation.