A GNN Based Approach to LTL Model Checking

Prasita Mukherjee and Tiark Rompf

Purdue University, West Lafayette IN 47906, USA,
{mukher39,tiark}@purdue.edu

Abstract. Model Checking is widely applied in verifying complicated and especially concurrent systems. Despite of its popularity, model checking suffers from the state space explosion problem that restricts it from being applied to certain systems, or specifications. Many works have been proposed in the past to address the state space explosion problem, and they have achieved some success, but the inherent complexity still remains an obstacle for purely symbolic approaches. In this paper, we propose a Graph Neural Network (GNN) based approach for model checking, where the model is expressed using a Büchi automaton and the property to be verified is expressed using Linear Temporal Logic (LTL). We express the model as a GNN, and propose a novel node embedding framework that encodes the LTL property and characteristics of the model. We reduce the LTL model checking problem to a graph classification problem, where there are two classes, 1 (if the model satisfies the specification) and 0 (if the model does not satisfy the specification). The experimental results show that our framework is up to 17 times faster than state-of-the-art tools. Our approach is particularly useful when dealing with very large LTL formulae and small to moderate sized models.

Keywords: Model Checking · Graph Neural Network · Graph Classification.

1 Introduction

Model checking, proposed by Clarke et al. [9] is defined as the problem of deciding whether a property holds for all executions of a system. A property is typically specified in a temporal logic like Linear Temporal Logic (LTL), Computational Tree Logic (CTL), etc. Model checking has proven to be extremely effective in verifying complex systems against a set of specifications. It has gained wide acceptance in the fields of systems engineering [3], protocol verification in hardware and software systems, malware detection and security protocols [1].

Generally, formal specifications are expressed using temporal logic formulae like LTL, CTL, etc. The system/model is expressed using automata like Büchi [8], Muller, Kripke structures [22], or, Petri nets [20] to express concurrent systems. Given the system and the specification expressed using temporal logic, model
checking can automatically verify whether the system satisfies the specification. The result of model checking on a system expressed using Büchi automaton will be true if the automaton satisfies the specification expressed using temporal logic, and false if the automaton does not satisfy the specification.

Linear Temporal Logic (LTL) [21] and Computational Tree Logic (CTL) [14], [5] are the two most widely used temporal logics to express a specification both in industry and academia. However, LTL-based model checking suffers from the state space explosion problem [26] that hinders the performance of a model checker, especially with respect to very large LTL formulae or very large models. Methods like partial order reduction [16], symmetry [10], [11], bounded model checking [6], [17], equivalence relations etc. have been proposed to address the state space explosion problem. These methods have achieved some success, but in general the problem remains hard and hence the state space explosion problem still constitutes a major bottleneck in many applications of model checking.

In recent years, machine learning techniques have gained success in dealing with the state space explosion problem. Zhu et al. [29] proposed model checking based on binary classification of machine learning. They represented the system using Kripke structures and specification using LTL formulae. The Kripke structures and LTL formulae served as input features to the machine learning algorithms. A “0/1” label served as the supervised flag during training, where the value 1 stated that the Kripke structure accepts the LTL formula and 0 otherwise. They evaluated the performance of their framework on machine learning algorithms like Random Forest, Decision Trees, Boosted Tree and Logistic Regression. They achieve similar accuracy as the state-of-the-art algorithms used in classical LTL model checking and partially avoid the state space explosion problem. However, their method does not generalize well when tested with formulae of varying lengths in the same dataset. Behjati et al. [4] proposed a reinforcement learning based approach for on-the-fly model checking for properties expressed with LTL formulae. Their primary focus is on looking for invalid runs or counterexamples by awarding heuristics with a reinforcement learning agent. Their experimental results conclude that their approach performs much better than the classical model checking algorithms and can verify models with very large state spaces, but the termination condition of the agent searching the state space is still bounded.

In this paper, we propose a Graph neural network [23] based framework to perform LTL-based model checking. The model of the system is expressed using a Büchi automaton and the specification is expressed with an LTL formula. In this paper, we classify whether the Büchi automaton of a system satisfies the LTL formula with the help of Graph neural network based techniques. Encoding a Büchi automaton as a Graph neural network (GNN) and the LTL formula as a feature of the GNN along with a “0/1” label during training and a “0/1” expected output during testing reduces the problem of LTL-based model checking to a graph classification problem. This prevents us from encountering the famous state space explosion problem when it comes to polynomial sized models and
very large LTL formulae, and hence predicts results significantly faster, especially for very large LTL formulae.

The experimental results show that our framework achieves moderate to high accuracy on training, validation and test datasets. The accuracy of our model is also not negatively affected by the varying length LTL formulae and models present in the same dataset. These results indicate that our framework is independent of the length of the LTL formulae and models. The model also predicts results much faster than the state-of-the-art model checkers, especially on very large LTL formula, for which the model checkers take hours to solve.

The remainder of the paper is organized as follows:

– In section 2, we review the basics of model checking and graph neural networks.
– In section 3, we present our embedding approach in detail.
– In section 4, we present the experimental results which describe the efficiency of our framework with respect to accuracy and time taken to determine results in comparison to the state-of-the-art tools.
– In section 5, we describe some of the related works that use machine learning based techniques and compare our proposed approach with them.
– In section 6, we describe the limitations of our work
– In section 7, we draw conclusions and describe possible future work.

2 Background

2.1 Büchi Automata

In automata theory, a Büchi automaton is a model that either accepts or rejects inputs of infinite length. The automaton is represented by a set of states (some of which are initial, some final, and others neither initial nor final), a transition relation, which determines which state should the present state move to, depending on the alphabet. The model accepts an input if and only if it visits at least one accepting state infinitely often for the input. A Büchi automaton can be deterministic or non-deterministic. We deal with non-deterministic Büchi automata models in this paper, as they are strictly more powerful than deterministic Büchi automata models.

A non-deterministic Büchi automaton $B$ is formally defined as the tuple $(Q, q_0, \sum, \Delta, q_f)$ where the components are:

– $Q$ is the set of all states of $B$. $Q$ is finite.
– $q_0$ is the set of all initial states
– $\sum$ is the finite set of alphabets
– $\Delta : Q \times \sum \rightarrow 2^Q$ is the transition relation that can map a state to a set of states on the same input alphabet
– $q_f$ is the set of final states
2.2 LTL Based Model Checking

Linear Temporal Logic (LTL), is a type of temporal logic that models properties with respect to time. An LTL formula is constructed from a finite set of atomic propositions, logical operators not (\(\neg\)), and (\(\land\)), and or (\(\lor\)), and temporal operators:

- \(\mathbf{X}\) for next
- \(\mathbf{U}\) for until
- \(\mathbf{W}\) for weak until
- \(\mathbf{F}\) for eventually
- \(\mathbf{G}\) for globally
- \(\mathbf{R}\) for release, and,
- \(\mathbf{W}\) for weak until

Additional logical operators like implies (\(\rightarrow\)), equivalence (\(\leftrightarrow\)), etc. are expressed using the operators (\(\neg\), \(\land\), \(\lor\)).

2.3 LTL Based Model Checking

Given a system \(M\) expressed by a Büchi automaton \(B_M\) and a specification \(\phi\) expressed by an LTL formula, the model checking problem decides whether the system \(M\) satisfies the specification \(\phi\). The problem is solved by computing the Büchi automaton for the negation of the specification \(\phi\), i.e., \(B_{\neg \phi}\), followed by the product of \(B_M\) and \(B_{\neg \phi}\). The problem then reduces to checking the language emptiness problem of the product Büchi automaton \(B = B_M \times B_{\neg \phi}\). The language accepted by \(B\) is said to be empty if and only if the automaton rejects all inputs, i.e., it does not pass through any accepting state infinitely often for every input.

Construction of \(B_M\) is linear in the size of \(M\), \(B_{\neg \phi}\) is exponential in the size of \(\neg \phi\). The product construction would also lead to an automaton of size \(|M| \times 2^{|\neg \phi|}\), which can blow up for very large LTL formulae. The proposed approach prevents the explicit construction of \(B_{\neg \phi}\) and the product \(B\) by incorporating the LTL formulae as node features of the Graph neural network.

2.4 Graph Neural Networks

Graph Neural Networks (GNN) extend regular neural networks to better support data processing in graph domains. A GNN is represented by a graph \(G = (V, E)\), where the set \(V\) represents the nodes and the set \(E\), the edges of the graph. The graph can be directed or undirected. Every node \(i \in V\) is represented by an initial embedding \(h_i^0 \in \mathbb{R}^d\). Edges \(e \in E\) can also have features called edge attributes.

The target of a GNN is to learn node embeddings, \(h_i \in \mathbb{R}^d\), \(\forall i \in V\) which contain the neighbourhood information for the node \(i\). The embedding is a \(d\) dimensional vector representation for a node \(i\) that stores the final output label of every node after the execution of the model. GNNs capture the embeddings
via message passing between neighboring nodes in the graph. In contrast to standard neural networks, GNNs have the capacity to retain the node embeddings captured by message passing up to any depth. GNNs have achieved groundbreaking success in graph representation data, especially in node classification and graph classification tasks. This motivated us to represent $B_M$ as a GNN and map it to a graph classification problem, where the task is to classify the GNN representation of $B_M$ based on whether it satisfies (1) the LTL formula $\phi$ or not (0).

The comparative study by Errica et al. [15] on GNNs for graph classification showed that GIN (a variant of GNN) showed very promising results when it came to graph classification. Hence, in this paper, we use GIN [27]. The message passing framework of GIN is described as follows:

$$h_i^{(l+1)} = f_\Theta((1 + \epsilon)h_i^l + \text{aggregate}({h_j^l, j \in N(i)})))$$

where,

- $h_i^l$ is the node embedding at layer $l$,
- $\epsilon$ is a trainable parameter,
- $N(i)$ is the neighbour set of node $i$,
- $\text{aggregate}$ states the aggregation method (maximum, mean, or, minimum),
- $f_\Theta$ denotes the activation function.

We describe the details of our encoding of the model $B_M$ and the specification $\phi$ in GIN in Section 3.

3 Proposed Method

The proposed method determines whether a model, represented by a Büchi automaton $B_M$ satisfies a specification $\phi$ represented by an LTL formula. We model the problem as a supervised learning problem on Graph neural networks (GNN), where a single bit (0/1) of supervision is provided along with $B_M$ and $\phi$ during training. Here, bit 1 means that $B_M$ satisfies $\phi$ and 0 otherwise.

We represent $B_M$ as a graph $G = (V, E)$, where the nodes of $G$ are the states of $B_M$ and the edges of $G$ are the transitions of $B_M$. $G$ is directed in nature. We represent the LTL formula $\phi$ as a feature of the GNN. We construct the LTL formula from the alphabet set $\{a,...,z\}$, operators $\{F, G, X, !, U, W, R, M, \&, \neg\}$, and a special value 1 that indicates true.

We encode the LTL formula $\phi$ as follows:

- We assign unique numerical values to each alphabet and operator
- We construct the expression tree corresponding to $\phi$
- We construct the encoding by appending the numerical values of the nodes (which are alphabets or operators, or 1) corresponding to the inorder traversal of the expression tree
The edges of $B_M$ also have labels from the alphabet set $\{1, a, ..., z\}$ and operator $\{!\}$. For example, an edge can have the label $\{a, !b\}$. An edge label 1 indicates true (i.e., every alphabet holds in that edge). We encode the edges as a binary vector of 53 elements, where the first element denotes the label 1, the next 26 elements denote labels $\{a, b, ..., z\}$ and the next 26 elements denote labels $\{!a, !b, ..., !z\}$. The edge encoding will contain 1 at the indices where the label holds and 0 everywhere else. For example, for the label $\{a, !b\}$, the edge encoding will contain 1 at indices 1 (for $a$) and 28 (for $!b$) and 0 everywhere else.

The node embedding $h_0^i$ (i.e., the initial embedding) for the node $i$ is defined as the concatenation of the node number, node type, and LTL encoding. The node number is defined as the vertex number of the graph. The node type consists of four classes (bit representation), namely:

- Initial only (10) - Here, the node is the initial state of the automaton
- Final only (01) - Here, the node is one of the final states of the automaton
- Both (11) - Here, the node is both initial and final state of the automaton
- Neither (00) - Here, the node is neither the initial, nor a final state of the automaton

The node embeddings are then updated according to the Equation 1 for every layer $l$. We are essentially performing a binary classification (i.e., we expect the output to be 0 or 1). Hence we map the node embeddings into two classes at the last layer. We describe the encoding of the specification $\phi$, edge features and node embeddings $h_0^i$ for every node $i$ in detail in the example below.

### 3.1 Proofs

**Lemma 1.** Every LTL formula has a unique encoding

**Proof.** By contradiction, let us assume that two different LTL formulae ($l_1$ and $l_2$) have the same encoding.

Let, $encl_1 = [n_1, n_2, ..., n_k]$ and $encl_2 = [n'_1, n'_2, ..., n'_k]$, where $encl_1$ represents the encoding for $l_1$, and $encl_2$ represents the encoding for $l_2$.

By our assumption, $l_1$ and $l_2$ have the same encoding, hence $n_1 = n'_1, n_2 = n'_2, ..., n_k = n'_k$.

Since, $l_1$ and $l_2$ are different formulae, their expression trees will be different. Since, the encoding corresponding to the LTL formula represents the inorder traversal corresponding to the expression tree, there would be at least an $i$, where $1 \leq i \leq k$, such that $n_i \neq n'_i$. Hence, we arrive at a contradiction. Thus, $encl_1 = encl_2$ is not true, where $l_1$ and $l_2$ are different formulae. Thus, every LTL formula has a unique encoding.

**Lemma 2.** Every $(B_i, L_i)$ has a unique representation

**Proof.** We prove the lemma by case analysis, by analyzing two different $(B_i, L_i)$ combinations, at a time. Here $(B_i, L_i)$ denotes the tuple where $B_i$ is the automaton and $L_i$ is the LTL formula.
Case a. \((B_k, L_k)\) and \((B_j, L_j)\)

In this case, \(B_k \neq B_j\) and \(L_k \neq L_j\). Since, \(L_k \neq L_j\), \(L_j\) and \(L_k\) would have different encodings from the above lemma. Hence, irrespective of the representation of \(B_k\) and \(B_j\), \((B_k, L_k)\) and \((B_j, L_j)\) would have different representations because \(L_k\) and \(L_j\) are different.

Case b \((B_k, L_k)\) and \((B_k, L_j)\)

In this case, \(L_k \neq L_j\). This case trivially follows from case (a). The only difference being \(B_k\), which is same across both tuples. Since, \(L_k \neq L_j\), the tuples will have different representations in spite of \(B_k\) being same.

Case c \((B_k, L_k)\) and \((B_j, L_k)\)

In this case, \(B_k \neq B_j\), and \(L_k\) is same across the tuples. This case can further be divided into two subcases, where we analyze \(B_k\) and \(B_j\):

Case c.a \(B_k\) and \(B_j\) have different number of nodes, or edges

If \(B_k\) and \(B_j\) have different number of nodes, the number of node embeddings for \(B_k\) will be different from that of \(B_j\). Hence, \((B_k, L_k)\) and \((B_j, L_k)\) will have different representations, as they will have different number of embeddings, even though the embeddings would contain the encoding of \(L_k\).

If \(B_k\) and \(B_j\) have the same number of nodes but, different number of edges. In this case, the number of node embeddings are same, but they differ by at least one edge. Since, edges also have embeddings and are a part of the message passing framework, the representations of \((B_k, L_k)\) and \((B_j, L_k)\) will still be different as they will vary in the number of edges, hence at least one edge embedding.

Case c.b \(B_k\) and \(B_j\) have same number of nodes and edges

This is the most complex part of the analysis, and can be further divided into two subcases, with examples that just paint a clearer picture of the subcases:

Case c.b.i \(B_k\) and \(B_j\) differ in at least one edge label

\[
\begin{array}{c}
q_0 \\
\{a, b\} \\
\rightarrow \\
\{q_0, q_1\} \\
\{a, !b\} \\
q_1
\end{array}
\]

Fig. 1. Automaton \(B_k\)

From figures [1] and [2] we see that \(B_k\) and \(B_j\) have the same number of nodes and edges, and differ in exactly one edge label, which represents the subcase under consideration. Although, the number of node and edge encodings will be same in this case, the content of at least one edge encoding will be different, due to the edge labels being different and the edge encodings being dependent on the edge labels. Hence, without loss of generality the representations of \((B_k, L_k)\) and \((B_j, L_k)\) will still be different.

Case c.b.ii \(B_k\) and \(B_j\) have the same edge labels
From figures 3 and 4, we see that $B_k$ and $B_j$ have the same number of nodes and edges, and have the same edge labels as well, which represents the subcase under consideration. Since, $B_k$ and $B_j$ are different automata models, but have the same number of nodes, same edges and labels, the only way they are different is the nature of the states. In this particular example, we see that for $B_k$, $q_0$ is both the initial and final state, and for $B_j$, $q_0$ is the initial state and $q_1$ is the final state.

In this case, even though the edge label encodings will be exactly the same, and the number of node embeddings will be the same, and each node embedding would contain the encoding of $L_k$, the node embeddings would still be different as we also include the type of state along with the identity of the node. Since, the nature of states would have to be different in this subcase in order for $B_k$ and $B_j$ to be different, the node embeddings of the states would differ in the representation of the nature of the state. Hence, in spite of the number of nodes, edges, and edge labels being same, we still get different encodings for $(B_k, L_k)$ and $(B_j, L_k)$ due to the nature of their states.

The above cases exhaustively cover all possible scenarios under which, two tuples $(B_j, L_j)$ and $(B_k, L_k)$ might have had the same representation, and on
analyzing every case, we come at the conclusion that they have different representations. Thus, every \((B_i, L_i)\) has a unique representation in our framework.

3.2 Example

LTL formula encoding Every LTL formula can be uniquely represented as a binary expression tree. Let us consider the LTL formula \((aUb)\&c\). Figure 5 represents the expression tree corresponding to this formula.

![Expression tree for the LTL formula \((aUb)\&c\)](image)

The expression tree is constructed as follows:

- The LTL formula is first converted into its postfix notation
- The postfix notion of the formula is used to construct the syntax tree
- The priorities of the operators for converting the formula into postfix are as follows:
  - \{!, G, F, X\} have the highest priority
  - \{U, R, W, M\} have the second highest priority
  - \{\&,\} have the least priority

In Figure 5, the operator nodes are represented by square shape and alphabet nodes are represented in circular shape. As an example, the alphabet \(a\) is assigned the number 0.02, \(b\) 0.03, \(c\) 0.04, \(U\) 0.32, and \& 0.36. The list obtained from the inorder traversal of this tree will be: \{0.02, 0.32, 0.03, 0.36, 0.04, 0, 0,..., 0\}. The vector has a lot of trailing zeros as our LTL representation vector is fixed at a constant size of 600, which is the maximum length of the LTL formula in our dataset. Hence, a formula with length less than 600 will have trailing 0s at the end.

Edge feature encoding Let us consider the Büchi automaton below:

Here \(q_0\) is the initial state and \(q_3\) is the final state. While all the edges have labels, we have explicitly shown the edge label \(\{a, !b\}\) corresponding to the edge \((q_1, q_0)\) and \(\{a, b\}\) corresponding to the edge \((q_0, q_2)\) for simplicity, just to show the edge encoding for a particular edge, and node embedding for all the nodes.

From the description of our approach, we know that every edge encoding is a vector of 53 elements. The edge feature encoding for the label \(\{a, !b\}\) will thus be: \{0,1,0,0,...,0,1,0,0,...,0\}.
Node embeddings} We describe the node embeddings for the initial layer, i.e., layer 0 (h₀). Following the definition of node embedding, the embedding for nodes are defined as:

- q₀ = {0, 1, 0, ltl_encoding}. Here q₀ is the initial state.
- q₁ = {1, 0, 0, ltl_encoding}. Here q₁ is neither the initial, nor a final state.
- q₂ = {2, 0, 0, ltl_encoding}. Here q₂ is neither the initial, nor a final state.
- q₃ = {3, 0, 1, ltl_encoding}. Here q₃ is a final state.

For a GNN to work, we need to assign all node embeddings of a graph to the same length. Hence, we assign the total length of a node embedding vector as the maximum length of the LTL formula in the dataset plus 3. This would lead to a string of zeros at the end for the automaton, LTL pair where the LTL formula size is smaller than the maximum length formula in the dataset.

4 Experiments

4.1 Model Hyperparameters

We trained the model with five convolutional layers. The node embeddings are initially encoded as a vector of size 603, and the edge encodings are a vector of length 53. The size of the LTL formulae vary between 1 to 600 and the number of states of the automata vary between 1 to 3756 states respectively.

We use the Adam optimizer [19] with learning rate 1e-3 and L2 regularization to prevent overfitting with weight decay 1e-3. We also use a dropout rate of 0.3, batch size of 1000 and ReLU as the activation function, and mean pooling as the pooling method. We train our model for a maximum of 2000 epochs and use the cross entropy loss as the loss function. We also incorporate an early stopping strategy, where the training stops if the training loss decreases below 0.1.

4.2 Data Generation

We train and evaluate our framework with LTL formulae generated from the tool Spot [13] and automata models from the tool LTL3BA [2]. We generate
5000 LTL formulae using the randltl feature of Spot and then feed the generated formulae as input to LTL3BA to get the required automata models.

We evaluate our framework on a dataset formed from the above, that comprises of 10000 pairs of LTL formulae $\phi$ and Büchi automaton $B_M$, with 8000 of them forming the training set, 1000 forming the validation set, and 1000 accounting for the test set. To test the validity of our model, we train and test it on a balanced dataset, i.e., we pick $(\phi, B_M)$ combinations such that half of them are satisfied and half are not. We call this a 50-50 split. We form 10000 total pairs by picking a $B_M$ from the above 5000 models, and pair it with the $\phi$ from which $B_M$ was generated (a satisfying pair, also called a positive sample) and another $\phi$ which generates a different automaton than $B_M$ (an unsatisfying pair, also called a negative sample). The ratio of yes to no is uniformly maintained between training, validation and test sets. The formed dataset has approximately 500 formulae of length greater than 250 and less than 500, 4000 formulae of length greater than 100 and less than 250, and rest of length less than 100. This dataset is a collection of models and LTL formulae, neither of which contain any personal information nor any offensive content, such as models describing harmful impact on society or, formulae containing the author’s names. We made sure that the randltl feature of Spot used to generate the LTL formulae did not take as variable names any author’s name, and manually checked whether the models and formulae describe any offensive scenario.

4.3 Results

We evaluate our model on a x86_64 GNU/Linux machine with CUDA 11.2 and GPU Persistence-M. We present the results and accuracy below and then compare our approach with the state-of-the-art tools Spot and LTL3BA, with respect to time and accuracy.

**Accuracy** From Table 1 we observe that our framework achieves at least 94% accuracy across training, validation and test datasets when the set of unique LTL formulae considered in the negative pairs are very small. For unique negatives as 1, even though we still have a 50-50 split, but the negative set comprises of all $(\phi, B_M)$ such that $\phi$ remains constant whereas $B_M$ varies from 1 to 5000. As the number unique negatives considered increases, the validation and test accuracy falls as well, and the loss curves also deviate way early. We still get decent results of 75% validation and test accuracy when the number of unique negatives are less than 1000. After 2000 unique negatives, the model gives accuracy of 55-65%, from which we can conclude that the present framework does not generalize well when we consider a large number of unique negative LTL formulae.

The loss curves show that both train (blue curve) and validation (orange curve) losses eventually becomes almost zero around 150 epochs when the number of unique negatives are less, but deviate as we keep on increasing the number of unique $\phi$’s in the negative set. The point of deviation also decreases as we expand the unique $\phi$’s. From these observations, we can infer that the model does
fit and generalize very well across different LTL formulae and models of varying length, when we don’t have much divergence in the negative set, in terms of the unique LTL formulae involved.

| Training | Validation | Test | Unique Negatives |
|----------|------------|------|------------------|
| 98       | 98         | 98   | 1                |
| 97       | 97         | 96   | 10               |
| 95       | 94         | 94   | 100              |
| 97       | 75         | 75   | 500              |
| 98       | 67         | 65   | 1000             |
| 96       | 60         | 60   | 2500             |
| 92       | 55         | 57   | 5000             |

Table 1. Approximate Accuracy (%) results across training, validation and test splits

Fig. 7. Training and validation loss curve

Comparison with Spot and LTL3BA  Spot is a tool that is widely used to generate LTL formulae, different types of automata on languages on infinite length words like, Büchi, generalized Büchi, Rabin, Parity, etc. Both Spot and LTL3BA can map an LTL formula to its corresponding Büchi automaton, and can also check the equivalence between two generated automata. LTL3BA generates a Büchi automaton approximately 4 times faster than Spot. Hence we
Fig. 8. Training and validation loss curve for 10 unique negatives

Fig. 9. Training and validation loss curve for 100 unique negatives
**Fig. 10.** Training and validation loss curve for 500 unique negatives

**Fig. 11.** Training and validation loss curve for 1000 unique negatives
Fig. 12. Training and validation loss curve

Fig. 13. Training and validation loss curve
generate the LTL formulae using Spot and map the formulae to their corresponding automata using LTL3BA.

We compare our framework against LTL3BA for the tuple \((\phi, B_M)\). We use LTL3BA to determine whether \(B_M\) satisfies \(\phi\), by taking \(B_{M'}\) as the automaton corresponding to the LTL formula \(\phi\), and the model \(B_M\) as the second, and checking whether the automata models are exactly the same. Checking whether the two models are same can be done by enumerating their states and transitions with respect to the input alphabets, and comparing them for equality. Hence, the primary challenge for LTL3BA is to generate \(B_{M'}\) corresponding to \(\phi\). Our framework does not perform too well in terms of accuracy across an extremely diverse dataset, and LTL3BA always reports correct results (hence has 100% accuracy), but our framework is much faster in comparison to LTL3BA, especially while dealing with large LTL formulae and corresponding complex models.

Our framework takes approximately 7 minutes to model check 10000 pairs of \((\phi, B_M)\). On the other hand, LTL3BA takes around 2 hours to model check one pair, where the size of the LTL formula varies from 300 to 500, and around 1 to 1.5 hours, where the size of the LTL formula varies from 150 to 300. For formulae of length less than 50, LTL3BA takes around 50ms to model check a pair. Hence our framework is up to 17 times faster on a per-pair basis, and thus is much faster than the state-of-the-art model checkers especially when it comes to large sized formulae and models.

5 Related Work

To the best of our knowledge, this is the first work that applies GNN based techniques to solve LTL-based model checking. Previously GNN based techniques have been proposed to solve SAT for boolean satisfiability [24] and satisfiability of 2QBF formulae [28]. However, satisfiability is not the same as model checking, as in the latter case we deal with two components, one being the model and the latter being the specification which is typically expressed as a temporal property.

Machine learning based techniques have also been used to select the most suitable model checker for an appropriate property and program [25]. This is a fundamentally different problem from ours as they focus on selecting an appropriate model checker, which is expected to give the most optimal performance for the give property and program. Another work by [7] attempts to reshape a model to satisfy the property. This is also different from our approach as we do not modify a model or a property to obtain a satisfiable result.

Hahn et. al [18] propose a novel approach to teach transformers to learn temporal properties. They train the transformer with different LTL formulae and tries to predict a trace satisfying the LTL formula under consideration. They present their results with respect to both syntax and semantic accuracy. Their results demonstrate that transformers can indeed learn a temporal specification and deduce semantically correct traces satisfying the formula under consideration in most cases. This work differs from our work in the sense that they predict a satisfiable trace, whereas we predict whether a model satisfies a specification,
which is expressed as an LTL formula. Their framework also work for relatively smaller length formulae in comparison to ours.

A reinforcement learning based approach has been proposed for on-the-fly model checking for properties expressed with LTL formulae. This tackles the model checking problem through learning, but is on-the-fly and searches for counterexamples with a reinforcement learning agent. Since this is a counterexample based approach, the termination condition of the agent searching the state space is still bounded. Hence, a definite answer may not be found. Our approach on the other hand always outputs an answer with good accuracy and in negligible time for one model and specification.

Another supervised learning based framework encodes Kripke structures as the model and LTL formula as the specification, and a flag bit specifying whether the model satisfies the specification. This framework is evaluated on a set of machine learning algorithms like logistic regression, random forest, etc. This is very similar to our approach, but their work does not give good accuracy when there are different length formulae and models in the same dataset. Hence, they require the dataset to be extremely restrictive as only same length formulae and same length models are allowed in a dataset. Hence, we need a large number of datasets (one for each unique length of LTL formula and model) and a large number of LTL formulae of same length to populate a dataset. Our approach does not suffer from these problems.

6 Limitations

From the experimental section, we observed and concluded that our framework is up to 17 times faster than the state of the art tools, but does not generalize well in diverse datasets, specifically datasets where no two tuples \((\phi_1, B_{M1})\) and \((\phi_1, B_{M2})\) exist for label 0, such that \(\phi_1 = \phi_2\). This is definitely a major drawback when it comes to trusting the model for safety critical systems and also addressing the question of generalizability.

7 Conclusions and Future Work

The proposed method avoids the non polynomial complexity of mapping the LTL formula \(\neg\phi\) into a Büchi automaton \(B_{\neg\phi}\), computing the product \(B = B_{\neg\phi} \times B_M\), which represents the intersection of \(B_M\) and \(B_{\neg\phi}\), and then checking emptiness of the obtained model \(B\). The complexity of the proposed method is hence polynomial in the size of \(|B_M|\).

The method is thus effective when the size of the formula \(\phi\) is huge and the size of the model \(B_M\) is small to moderate. However, computing \(B_M\) itself can be very inefficient when the size of the model is huge. Since we represent \(B_M\) as a graph to the GNN, we have to enumerate the states and transitions of \(B_M\) to construct the embeddings for training, validation and testing. Hence, the state space explosion problem is only avoided partially with this approach.
In future work, we would like to address the limitations of the present framework and aim to make the framework more generalizable, and hence more accessible for practical use. We would like to represent the LTL formulae as an encoder only transformer instead of an expression tree, as transformers can model long range dependencies really well, and hence can be used to encode temporal properties. We also aim to propose a smarter message passing framework that captures the acceptance condition of a Büchi automaton, instead of simply aggregating the embeddings of the immediate neighbors. We hope to achieve an increase in accuracy across diverse datasets and hence address the concern of generalizability with the combination of a transformer encoder and a novel message passing framework.

References

[1] Alessandro Armando, Roberto Carbone, and Luca Compagna. “LTL model checking for security protocols”. In: Journal of Applied Non-Classical Logics 19.4 (2009), pp. 403–429. doi: 10.3166/jancl.19.403–429 eprint: https://doi.org/10.3166/jancl.19.403–429 url: https://doi.org/10.3166/jancl.19.403–429

[2] Tomás Babiak et al. “LTL to Büchi Automata Translation: Fast and More Deterministic”. In: Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Ed. by Cormac Flanagan and Barbara König. Vol. 7214. Lecture Notes in Computer Science. Springer, 2012, pp. 95–109. doi: 10.1007/978-3-642-28756-5_8 url: https://doi.org/10.1007/978-3-642-28756-5_8

[3] Jiri Barnat et al. “Designing fast LTL model checking algorithms for many-core GPUs”. In: J. Parallel Distributed Comput. 72.9 (2012), pp. 1083–1097. doi: 10.1016/j.jpdc.2011.10.015 url: https://doi.org/10.1016/j.jpdc.2011.10.015

[4] Razieh Behjati, Marjan Sirjani, and Majid Nili Ahmadabadi. “Bounded Rational Search for On-the-Fly Model Checking of LTL Properties”. In: Fundamentals of Software Engineering. Ed. by Farhad Arbab and Marjan Sirjani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 292–307. isbn: 978-3-642-11623-0.

[5] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. “The Temporal Logic of Branching Time”. In: Acta Informatica 20 (1983), pp. 207–226. doi: 10.1007/BF01257083 url: https://doi.org/10.1007/BF01257083

[6] Armin Biere et al. “Symbolic model checking without BDDs”. In: b. Springer-Verlag, 1999, pp. 193–207.

[7] Rafael V. Borges, Artur S. d’Avila Garcez, and Luis C. Lamb. “Integrating model verification and self-adaptation”. In: ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp,
[8] J. Richard Büchi. “On a Decision Method in Restricted Second Order Arithmetic”. In: The Collected Works of J. Richard Buchi. New York, NY: Springer New York, 1990, pp. 425-435. ISBN: 978-1-4613-8928-6. DOI: 10.1007/978-1-4613-8928-6_23 URL: https://doi.org/10.1007/978-1-4613-8928-6_23

[9] E.M. Clarke et al. Model Checking, second edition. Cyber Physical Systems Series. MIT Press, 2018. ISBN: 9780262349451. URL: https://books.google.com/books?id=qJl8DwAAQBAJ

[10] Edmund M. Clarke et al. “Exploiting Symmetry in Temporal Logic Model Checking”. In: Formal Methods Syst. Des. 9.1/2 (1996), pp. 77–104. DOI: 10.1007/BF00625969 URL: https://doi.org/10.1007/BF00625969

[11] Edmund M. Clarke et al. “Symmetry Reductions in Model Checking”. In: Computer Aided Verification, 10th International Conference, CAV ’98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings. Ed. by Alan J. Hu and Moshe Y. Vardi. Vol. 1427. Lecture Notes in Computer Science. Springer, 1998, pp. 147–158. DOI: 10.1007/BFb0028741 URL: https://doi.org/10.1007/BFb0028741

[12] Manh Tuan Do, Noseong Park, and Kijung Shin. Two-stage Training of Graph Neural Networks for Graph Classification. 2021. arXiv: 2011.05097 [cs.LG]

[13] Alexandre Duret-Lutz et al. “Spot 2.0 — a framework for LTL and ω-automata manipulation”. In: Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA’16). Vol. 9938. Lecture Notes in Computer Science. Springer, Oct. 2016, pp. 122–129. DOI: 10.1007/978-3-319-46520-3_8

[14] E. Allen Emerson and Edmund M. Clarke. “Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons”. In: Sci. Comput. Program. 2.3 (1982), pp. 241–266. DOI: 10.1016/0167-6423(83)90017-5 URL: https://doi.org/10.1016/0167-6423(83)90017-5

[15] Federico Errica et al. “A Fair Comparison of Graph Neural Networks for Graph Classification”. In: CoRR abs/1912.09893 (2019). arXiv: 1912.09893 URL: http://arxiv.org/abs/1912.09893

[16] Patrice Godefroid. “Using Partial Orders to Improve Automatic Verification Methods”. In: CAV ’90: Proceedings of the 2nd International Workshop on Computer Aided Verification. London, UK: Springer-Verlag, 1991, pp. 176–185. ISBN: 3-540-54477-1.

[17] Viktor Gyuris and A. Prasad Sistla. “On-the-Fly Model Checking Under Fairness That Exploits Symmetry”. In: Computer Aided Verification, 9th International Conference, CAV ’97, Haifa, Israel, June 22-25, 1997, Proceedings. Ed. by Orna Grumberg. Vol. 1254. Lecture Notes in Computer Science. Springer, 1997, pp. 232–243. DOI: 10.1007/3-540-63166-6\_24 URL: https://doi.org/10.1007/3-540-63166-6\_24
[18] Christopher Hahn et al. “Teaching Temporal Logics to Neural Networks”. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, 2021. URL: https://openreview.net/forum?id=d0cQK-f4byz

[19] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980

[20] C. A. Petri. “Some Personal Views of Net Theory”. In: Applications and Theory of Petri Nets, Selected Papers from the 3rd European Workshop on Applications and Theory of Petri Nets, Varenna, Italy, September 27-30, 1982. Ed. by Anastasia Pagnoni and Grzegorz Rozenberg. Vol. 66. Informatik-Fachberichte. Springer, 1982, pp. 1–13. DOI: 10.1007/978-3-642-69028-0_1 URL: https://doi.org/10.1007/978-3-642-69028-0_1

[21] Amir Pnueli. “The Temporal Logic of Programs”. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 URL: https://doi.org/10.1109/SFCS.1977.32

[22] A Saul. “Kripke. Semantical analysis of modal logic I: Normal modal propositional calculi”. In: Zeitschrift für mathematische Logik und Grundlagen der Mathematik 9.5-6 (1963), pp. 67–96.

[23] F. Scarselli et al. “The Graph Neural Network Model”. In: IEEE Transactions on Neural Networks 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605

[24] Daniel Selsam et al. Learning a SAT Solver from Single-Bit Supervision. 2019. arXiv:1802.03685 [cs.AI]

[25] Varun Tulsian et al. “MUX: algorithm selection for software model checkers”. In: 11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India. Ed. by Premkumar T. Devanbu, Sung Kim, and Martin Pinzger. ACM, 2014, pp. 132–141. DOI: 10.1145/2597073.2597080 URL: https://doi.org/10.1145/2597073.2597080

[26] Antti Valmari. “A stubborn attack on state explosion”. English. In: Formal Methods in System Design 1.4 (1992). Project code: TK00018, pp. 297–322. ISSN: 0925-9856. DOI: 10.1007/BF00709154

[27] Keyulu Xu et al. “How Powerful are Graph Neural Networks?” In: CoRR abs/1810.00826 (2018). arXiv: 1810.00826 URL: http://arxiv.org/abs/1810.00826

[28] Zhanfu Yang et al. “Graph Neural Reasoning for 2-Quantified Boolean Formula Solvers”. In: Workshop on Learning and Reasoning with Graph-Structured Representations (co-located with ICML) (2019).
[29] W. Zhu, H. Wu, and M. Deng. “LTL Model Checking Based on Binary Classification of Machine Learning”. In: *IEEE Access* 7 (2019), pp. 135703–135719. DOI: [10.1109/ACCESS.2019.2942762].