OCTAL: Graph Representation Learning for LTL Model Checking

Prasita Mukherjee, Haoteng Yin
Department of Computer Science, Purdue University
West Lafayette, USA
{mukher39, yinh1t}@purdue.edu

ABSTRACT
Model Checking is widely applied in verifying the correctness of complex and concurrent systems against a specification. Pure symbolic approaches while popular, suffer from the state space explosion problem due to cross product operations required that make them prohibitively expensive for large-scale systems and/or specifications. In this paper, we propose to use graph representation learning (GRL) for solving linear temporal logic (LTL) model checking, where the system and the specification are expressed by a Büchi automaton and an LTL formula, respectively. A novel GRL-based framework OCTAL, is designed to learn the representation of the graph-structured system and specification, which reduces the model checking problem to binary classification. Empirical experiments on two model checking scenarios show that OCTAL achieves promising accuracy, with up to $11\times$ overall speedup against canonical SOTA model checkers and $31\times$ for satisfiability checking alone.

KEYWORDS
Model Checking (MC), Graph Neural Networks, Representation Learning

ACM Reference Format:
Prasita Mukherjee, Haoteng Yin. 2023. OCTAL: Graph Representation Learning for LTL Model Checking. In Proceedings of The Tenth International Workshop on Deep Learning on Graphs (DLG-KDD’23). ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION
Model checking [9] is defined as the problem of deciding whether a specification holds for all executions of a system. Generally, formal specifications are expressed using temporal logic formulae like LTL [25], CTL [25], etc. The system/model is expressed using automata like Büchi [7], Muller, Kripke structures [28], or Petri nets [24] to express concurrent systems. Given the system $B$ and specification $\phi$, model checking can automatically verify whether the system satisfies the specification by computing automaton $B_{\neg \phi}$, followed by the cross product of $B$ and $B_{\neg \phi}$, and then checks for the emptiness of the product (refer to Figure 1(e)). However, this approach suffers from the state space explosion problem [31], which severely hinders the performance of a model checker. Methods such as partial order reduction [16], symmetry [10], bounded model checking [4], have been proposed to address this problem, but it remains hard in general and constitutes a major bottleneck in deploying model checking for real-world applications.

Recently, machine learning (ML) methods [3, 32] have gained success in symbolic model checking, giving results in cases where traditional model checkers (MCs) time out. This makes them useful in scenarios where traditional MCs time out/fail to solve, with speed and efficiency being the key factors. For instance, in software verification, traditional MCs are not always a viable choice due to their high computational cost. Consider a large software development effort, where it would be favorable to use MC to ascertain a higher degree of correctness than pure testing. In such large-scale deployments, classical MCs often take a prohibitively long time for verification, especially when the system/specification has an exponential state space. In this case, only ML-based MCs can provide a practical solution, which broadens the applicability of MCs by trading off some amount of accuracy guarantees for better running time and scalability, which is particularly promising for large systems and/or specifications.

In this work, we address Model Checking through representation learning. Due to the structural essence of the input, model checking can be naturally formulated into graph tasks. This motivates us to propose a novel graph representation learning (GRL) based framework, OCTAL, to tackle this challenging problem. In OCTAL, the system is expressed as a Büchi automaton $B$ (Figure 1(a)) and the specification with an LTL formula $\phi$ (Figure 1(c)). Then, OCTAL determines whether $B$ satisfies $\phi$ by reducing the problem to binary classification on the graph union of $B$ and $\phi$ (Figure 1(d)).

We performed extensive experiments on OCTAL, traditional MCs, and neural network baselines for two scenarios of LTL model checking, in terms of both accuracy and speed on four datasets: two constructed from open competition RERS19 [19] and two others specifically constructed for this project. Experimental results show that OCTAL consistently achieves ~90% precision, recall, and accuracy indicating its generalization ability on unseen data, on varied length specifications, and its high utility in practice. In general, OCTAL is up to $11\times$ faster than the state-of-the-art (SOTA) traditional MCs, and achieves at least $31\times$ speedup in terms of satisfiability checking alone. Our major contributions can be summarized as follows: 1) LTL model checking is firstly formulated as a representation learning task, where $B$ and $\phi$ are expressed as graph-structured data. 2) Four datasets are constructed for LTL model checking benchmark: SynthGen, RERSGen correspond to the traditional model checking scenario, and SynthSpec, RERSpec correspond to the special model checking scenario.
Table 1: Operands/Variables in specification φ and system B.

| Operands/Variables | Specification φ | System B |
|--------------------|-----------------|----------|
| Meaning            |                |          |
|                     | a to z         | 26       |
|                     | ¬a to ¬z       | 26       |
|                     | a & b          | 1        |
|                     | a & ¬b         | 1        |
|                     | ¬a & b         | 26       |
|                     | ¬a & ¬b        | 26       |
| Cardinality        | 26             | 26       | 1 |

2 OCTAL: LTL MODEL CHECKING VIA GRAPH REPRESENTATION LEARNING

OCTAL determines whether a system B (Büchi automaton) satisfies a specification φ (LTL formula) through their unified graph representation, which is summarized in Figure 1. We formulate the problem as supervised learning on graphs, where the inputs B, φ and label (‘0/1’) are provided during training. Here, ‘1’ indicates that B satisfies φ and ‘0’ otherwise. Appendix A provides detailed explanation on Büchi automaton and traditional LTL Model Checking with concrete examples.

2.1 Variables and Operators

The systems and specifications we deal with are constructed from the operands/variables A, operators O = {G, F, R, W, M, X, U, !, &} and special variables true(I) and false(N), the specifics of which are described in Table 1 above and Table 7 in Appendix A. Each variable and operator has a distinct meaning and share across B and φ. A variable has its true or negated form (noted as A or ¬A).

2.2 Representation of System and Specification

System Graph G. We represent B as a bipartite graph G = (VG, EG), where VG = Vₜ ∪ Vₘ and EG ⊆ Vₜ × Vₘ. Here, Vₜ are the states of B, and Vₘ are the transitions of B. There is an edge between vₜ ∈ Vₜ and vₘ ∈ Vₘ if and only if vₜ is a source or destination state of the transition vₘ in B. G is undirected and the nature of vₘ being a source or destination vertex is captured in its node encoding.

Figures 1(a) and 1(b) illustrate B and the corresponding bipartite graph G. The two states q₀ and qₜ form the set Vₜ, and the transitions E₁, E₂, and E₃ form the set Vₘ. Since q₀ is a source and destination state for E₁, and a source state for E₃, there is an edge between q₀ and E₁, and q₀ and E₃ respectively. This is analogously followed by the rest of the graph. The intuition behind representing B as a bipartite graph is to capture the transition labels. Since we aim to learn the overall representation of a given system, both states and transitions in B play an essential role here. A state transitions into another state if and only if the transition label is satisfied. To learn the semantics of transitions and their corresponding labels, we map transitions as nodes as shown in Figure 1(b) accordingly, and therefore can obtain the representation for them, which is a function of the labels pertaining to the transition.

Specification Graph T. Every LTL formula can be represented as an expression tree T = (Ṽₜ, Eₜ) (see Figure 1(c)), which is constructed based on the precedence and associativity of the operators in LTL formulae, described as follows: 1) φ is converted to its postfix form, which is used to construct the expression tree; 2) The operators exhibit right associativity, where the unary operators (!, G, F, X) have the highest priority. 3) The binary temporal operators {U, R, W, M} have the second highest priority, and the boolean connectives {& |} have the lowest priority.

φT constitutes the operators and operands of φ, and Eₜ ⊆ Ṽₜ × Ṽₜ. φ is represented in Negation Normal Form (NNF) [11], which would place ! only before the operands. This allows us to represent !a as a variable in A and eliminate the ! operator. For example, the NNF equivalent of !(a U b) is !a R !b. Here, the ! operator is present only before a and b in the NNF equivalent of the formula !(a U b). Another compelling reason for representing φ in NNF is that transitions of B comprise of labels in true(1), A, and ¬A. Representing negation only before variables and eliminating the ! operator from φ enables the shared representation for variables across B and φ. As a result, there is no semantic difference between ¬A and ¬¬A: ¬a ∈ ¬A may occur in a transition label of B and !a ∈ A may occur in a leaf node of φ, but both ¬a and !a signify that a does not hold.

2.3 Bridging System and Specification via Graph Union

To establish the relation between graphs of system G and specification T, we propose to construct the unified graph C to model them jointly. Traditional approaches of model checking computes the intersection of B and B₁φ by their cross product, which results in an automaton B’ whose states are the product of the states of B and B₁φ (Figure 1(e)), and its transitions depend on the transition labels of both B and B₁φ. Since our main goal is to avoid constructing B₁φ and thus B’, we directly feed the input graph without products to OCTAL and aim to use neural networks to learn the latent correspondence by combining G and T as a joint framework C in the following way. Each transition label consists of operands and variables in A or ¬A, which is shared across the system and specification. Based on this observation, we join graphs G and T by adding a link between the corresponding nodes Vₜ ∈ G and Ṽₜ ∈ T if they contain the same variable/operand that belongs

Figure 1: Illustration of system B, bipartite format of B in graph G, expression tree T for specification φ, the unified framework C approximating the cross product B’ = B × B₁φ, with polynomial complexity of |B + φ|, and B’ as computed by traditional MCs.
The key intuition here is to jointly learn the representation $E$ that is used as the non-linear activation between GNN and MLP. A dropout rate [14] of 0.1 is used, along with 1D three-layer GNN. Mean pooling is used to aggregate the learned embeddings. Part VI remains 0 source and destination vertex of $E$ as contained in $V_E$, and the second bit is true if $a \in \mathcal{A}$ or $\neg a \in \neg \mathcal{A}$.

### 2.4 Node Encoding and Learning Unified Graph

Each node $v \in C$ consisting of operands/variables is represented by a vector as follows

\[
\begin{bmatrix}
\{1, \ldots, 3\} \\
\vdots \\
\{7, \ldots, 9\}
\end{bmatrix}
\begin{bmatrix}
\phi_1 \\
\vdots \\
\phi_9
\end{bmatrix}
\]

Part I of 1 bit is reserved for the special variable $\texttt{true}(1)$ or $\texttt{false}(N)$. Part II encodes $\forall a \in \mathcal{A}$, with size of $|\mathcal{A}|$. Part III of size $|\mathcal{A}|$ encodes variables/operands in either $\neg \mathcal{A}$ or $\mathcal{A}$, as both of them are semantically equivalent. Part IV corresponds to operators in $O$ except $\neg$, with the size of $|O| - 1$. Part $V$ represents the type of state $q$ of $B$ in 2 bits, where the first bit is true if $q$ is an initial state, and the second bit is true if $q$ is a final state. Part $VI$ represents the source and destination vertex of $B$, as $G$ is undirected in nature. Parts I through $V$ use one-hot encoding for indication, and part $VI$ uses the source and destination vertex numbers. Part $VI$ remains 0 other than vertices $v \in V_E$, as they correspond to the transitions of $B$ which have a source and destination. The difference between the components in parts I through $V$ is thus captured by their positions in the vector.

GNNs as a powerful tool can capture both structural information and node features for graph-structured data through propagation and aggregation of information through message passing, which is ideal for exploiting the structural correspondence between the $B$ and $\phi$, in addition to the semantics of transitions. Graph Isomorphism Network (GIN, [20]), one of the most expressive GRL models, is employed to learn the representation of the unified framework $C$. The key intuition here is to jointly learn the representation $G$ and $T$. Since two structurally similar $B$’s (or $\phi$’s) can represent different behaviors depending on the contents of the transition, both structure and labels that describe the semantics of $B$ (or $\phi$) are equally important for LTL model checking. Modeling the system or the specification separately would lose the crucial connection between them, which is the essential component in traditional model checkers formed as graph products. Hence, we deploy GIN to capture both structure and semantics of $B$ and $\phi$ jointly in the representation learnt, and the significance of the joint representation for $C$ is further solidified in Sections 3.4 and 3.6.

### 3 EVALUATION

#### 3.1 Architecture and Hyperparameters

The architecture of OCTAL (Figure 3, Appendix B) comprises a three-layer GNN. Mean pooling is used to aggregate the learned node embeddings. A dropout rate [14] of 0.1 is used, along with 1D batch normalization [27] in every convolution layer of GNN. ReLU [1] is used as the non-linear activation between GNN and MLP layers. Every node in $C$ has an initial embedding of length 66. The GNN framework produces an embedding for $C$ of dimension 128 after mean pooling. MLPs take this hidden graph representation as the input and produce a ‘0/1’ result as the final prediction.

#### 3.2 Datasets

We present four datasets namely, SynthGen, SynthSpec, RERSGen and RERSSpec. The datasets constitute of datapoints of the form $(B, \phi, I)$, where $B$ is the system, $\phi$ is the specification and $I$ is the label. $I = 1’$ if $B$ satisfies $\phi$ and ‘0’ otherwise. SynthGen and RERSGen correspond to the general model checking scenario, while SynthSpec and RERSSpec correspond to the special model checking scenario. The datasets SynthGen (RERSGen) and SynthSpec (RERSSpec) share the same $B$ and $\phi$, differing in $I$. Hence, we refer to the distribution of SynthGen (SynthSpec) as Synth, and RERSGen (RERSSpec) as RERS. Synth is constructed synthetically and RERS is adopted from the specifications of the RERS model checking competition 2019. The motivation behind constructing the mentioned datasets are to test the accuracy, speedup and generalizability of OCTAL when it comes to complex, lengthy and varied length specifications and/or systems in the same dataset, which previous ML based works failed to solve. The purpose of specifically designing the synthetic dataset was to incorporate a diverse set of specifications, which is different from RERS as the latter caters to traditional model checking competitions where MCs can’t handle the length and complexity of specifications/systems beyond a threshold. The statistics and construction of the datasets are summarized in Table 2, and detailed in Appendix B.3.

#### 3.3 Experimental Settings

**Training.** OCTAL is trained with an 80-20 split between training and validation sets, which contain equal number of positive and negative samples for classification and are randomly shuffled. We use Adam [21] with initial learning rate $1r=1e^{-5}$, and the Binary Cross Entropy [26] as the loss function for all experiments. Early stopping is adopted when the highest accuracy on validation no longer increases for five consecutive checkpoints. All experiments are run 5 times independently, and the average performance and standard deviations for accuracy, precision, and recall are reported.

**Baselines.** Two classes of methods are selected to compare for LTL model checking:

*Traditional Model Checkers* LTL3BA, Spin [18] and Spot are the SOTA tools that perform traditional symbolic model checking. LTL3BA is the fastest tool among them to compute the SOTA tools that perform traditional symbolic model checking.

Table 2: Statistics of Synth and RERS.

| Dataset | Len_LTL | #State | #Transition |
|---------|---------|--------|-------------|
| Synth   | [1 - 80]| [1 - 95]| [1 - 1,711] |
| RERS    | [3 - 39]| [1 - 21]| [3 - 157]  |

We select LTL3BA and Spot as the baselines for the speed test due to their superior performance, and run it for every $(B, \phi)$. Since $B$ corresponds to a specification $\phi'$ as per our dataset description, to model check $(B, \phi)$ using LTL3BA, we provide the LTL formula $(\phi' \land \phi')$ as input to LTL3BA and verify whether the automaton generated is empty. To model check $(B, \phi)$ using Spot, we check whether $\phi'$ implies $\phi$ through its command line interface.
Table 4: Speedup for general LTL model checking. LT3BA (Spot) (I) represents inference-only speedup. LT3BA (Spot) (O) represents the sum of inference and graph construction overhead speedup.

| Models       | Accuracy | Precision | Recall |
|--------------|----------|-----------|--------|
| MLP          | 0.64/0.86 | 0.17/0.18 | 0.67/0.80 |
| LinkPredictor| 0.66/0.81 | 0.65/0.81 | 0.66/0.81 |
| OCTAL(GCN)   | 0.76/0.89 | 0.77/0.91 | 0.77/0.89 |
| OCTAL(GIN)   | 0.77/0.91 | 0.80/0.91 | 0.80/0.91 |

Table 5: Classification Accuracy, Precision and Recall for special model checking scenario for SynthSpec and RERSSpec

| Models       | SynthSpec | RERSSpec | SynthSpec | RERSSpec | SynthSpec | RERSSpec |
|--------------|-----------|----------|-----------|----------|-----------|----------|
| MLP          | 0.48/0.80 | 0.50/0.80 | 0.46/0.75 | 0.45/0.75 | 0.46/0.75 | 0.45/0.75 |
| LinkPred     | 0.51/0.80 | 0.51/0.80 | 0.51/0.80 | 0.51/0.80 | 0.51/0.80 | 0.51/0.80 |
| OCTAL(GCN)   | 0.91/0.80 | 0.92/0.80 | 0.91/0.80 | 0.91/0.80 | 0.91/0.80 | 0.91/0.80 |
| OCTAL(GIN)   | 0.93/0.80 | 0.93/0.80 | 0.93/0.80 | 0.93/0.80 | 0.93/0.80 | 0.93/0.80 |

Table 6: Speedups for special LTL model checking.

| Dataset      | LT3BA(I) | LT3BA(O) | Spot(I) | Spot(O) |
|--------------|----------|----------|---------|---------|
| SynthSpec    | 53.5×    | 53.5×    | 53.5×   | 53.5×   |
| RERSpec      | 54.5×    | 54.5×    | 54.5×   | 54.5×   |

3.5 Complexity Analysis

Table 4 shows the speedup of OCTAL with respect to LT3BA and Spot, for inference only, and inference with graph construction overhead. Compared to traditional MCs, with preprocessing overhead considered, NN-based models are still ~11× faster than LT3BA and ~1.7× faster than Spot for SynthGen. NN-based models are ~2× faster than LT3BA and Spot for RERSGen. It is worth noting that, in terms of inference alone, OCTAL is ~35×, ~54× faster than LT3BA and ~52×, ~31× faster than Spot on both datasets. Hence, we infer that OCTAL can outperform the SOTA traditional model checkers with respect to speed, along with consistent accuracy across different datasets. We also conclude that OCTAL tends to have an increase in speedups over traditional MCs when it comes to more complex and lengthy specifications/systems, as from the distribution, we see that Synth subsumes RERS. Note that, as a proof of concept, the graph preprocessing time presented above is not extensively optimized in terms of speed. We aim to provide a parallel graph construction algorithm in the future that would significantly reduce the preprocessing overhead.

Traditional model checkers may perform ~Nφ to B-φ, compute the product B′ = B-φ × B, and then check emptiness of B′. The use of the union operation with GRL framework enables OCTAL to avoid the non-polynomial complexity of graph product. Accordingly, the complexity of our proposed method is reduced to polynomial in the size of |B + φ|.

3.6 Case Study: On the equivalence of B and φ

We evaluate OCTAL on a special case of model checking, where B accepts φ if they are equivalent (Figure 2(b)). The goal of this setting is to evaluate OCTAL on system equivalence which is indeed a hard problem. Tables 5 and 6 show the accuracy and speedup results. Here too, the message passing networks outperform the MLP and LinkPredictor, signifying the importance of learning the union of G and T. The results for equivalence checking for OCTAL are very impressive. OCTAL consistently reports accuracy ~95% across SynthSpec and RERSSpec which is significantly higher than...
the general case. Further experiments on runtimes obtained for the datasets corresponding to general and special model checking is presented in Appendix C.

4 RELATED WORK
To the best of our knowledge, this is the first work that applies graph representation learning to solve LTL model checking. Previously, the most relevant work to us was using GNNs to solve SAT for boolean satisfiability [29], and automated proof search [23] in the higher-order logic space. Learning-based approaches have been used to select the most suitable model checker [30]. [6] attempts to learn how to reshape a system [8] to satisfy the property in temporal logic. [17] trains a transformer to predict a satisfiable trace for an LTL formula. [32] proposes ML-based model checking, with the system being Kripke structures and specification being LTL formulas, both serving as input features to supervised ML algorithms. Their framework is shown to perform well with formulae of the same lengths, otherwise not. [3] proposes a reinforcement learning based approach for on-the-fly LTL model checking, which is designed to look for invalid runs or counterexamples by awarding heuristics with an agent. Their approach performs faster than the classical model checkers and can verify systems with large state spaces, but the state space that an agent can reach is still bounded.

5 CONCLUSIONS AND FUTURE WORK
OCTAL is a novel graph representation learning based framework for LTL model checking. It can be extremely useful for the first line of the software development cycle, as it offers reasonable accuracy and robustness for early and quick verification, compared to time-consuming unit tests and other efforts in ensuring the correctness of a given system. OCTAL is not intended to replace traditional model checkers, it rather makes model checking affordable and scalable for scenarios where traditional model checkers are infeasible. It can also be enhanced with a guarantee provided by applying traditional model checkers to limited candidates filtered by OCTAL. In future, we propose to improve OCTAL on the false negatives for implication cases and extend OCTAL to support the generation of a counterexample trace for the ‘no’ answers. Since the counter-example generation is a relatively easier problem, tentatively, the user can invoke a traditional model checker to obtain it.

6 ACKNOWLEDGEMENTS
The authors sincerely thank Professor Tiark Rompf and Dr. Susheel Suresh for their valuable feedback throughout the project.

REFERENCES
[1] Abien Fred Agarap. 2018. Deep learning using rectified linear units (relu). arXiv preprint arXiv:1803.08375 (2018).
[2] Tomáš Babiák, Mojmír Krátký, Vojtech Rehák, and Jan Strejček. 2012. LTL to Buchi Automata Translation: Fast and More Deterministic. In Proceedings of the 18th International Conference of Tools and Algorithms for the Construction and Analysis of Systems, Vol. 7214. Springer, 95–109.
[3] Ranieh Behjati, Marjan Sirjani, and Majid Nili Ahmadabadi. 2010. Bounded Ratio- nal Search for On-the-Fly Model Checking of LTL Properties. In Fundamentals of Software Engineering. Springer Berlin Heidelberg.
[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symb- loic model checking without BDDs. In International conference on tools and algorithms for the construction and analysis of systems. Springer, 1–13.
[5] František Blahoudek, Alexander Duret-Lutz, Mojmír Krátký, and Jan Strejček. 2014. Is There a Best BüChi Automaton for Explicit Model Checking?. In Proceedings of the 2014 International SPIN Symposium on Model Checking of Software. 68–76.
[6] Rafael V. Borges, Artur S. d’Avila Garcez, and Luis C. Lamb. 2010. Integrating model verification and self-adaptation. In Proceedings of the 25th International Conference on Automated Software Engineering.
[7] J. Richard Buchi. 1990. On a Decision Method in Restricted Second Order Arithmetic. Springer New York, 425–435.
[8] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. 1999. NuSMV: A new symbolic model verifier. In Proceedings of the 11th International Conference on Computer Aided Verification. Springer, 495–499.
[9] E.M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith. 2018. Model Checking, second edition. MIT Press.
[10] Edmund M. Clarke, Somesh Jha, Reinhard Enders, and Thomas Fikorn. 1996. Exploiting Symmetry in Temporal Logic Model Checking. Formal Methods Syst. Des. 9, 1 (1996), 77–104.
[11] Adnan Darwiche. 2001. Decomposable negation normal form. Journal of the ACM (JACM) 48, 4 (2001), 608–647.
[12] Alexandre Duret-Lutz, Alexandre Lewkowicz, Ananyu Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. 2016. Spot 2.0—a framework for LTL and ω-automata manipulation. In Proceedings of the 14th International Sym- posium on Automated Technology for Verification and Analysis, Vol. 9958. Springer, 122–129.
[13] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. arXiv preprint arXiv:1903.02428 (2019).
[14] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In international conference on machine learning, PMLR, 1050–1059.
[15] M.W Gardner and S.R Dorling. 1998. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. Atmospheric Environment 32, 14 (1998), 2627–2636.
[16] Patrice Godefroid. 1991. Using Partial Orders to Improve Automatic Verification Methods. In Proceedings of the 2nd International Workshop on Computer Aided Verification. 176–185.
[17] Christopher Hahn, Fréderick Schmitt, Jens U. Kreber, Markus Norman Rabe, and Bernd Finkbeiner. 2021. Teaching Temporal Logics to Neural Networks. In International Conference on Learning Representations.
[18] Gerard J. Holzmann. 1997. The model checker SPIN. IEEE Transactions on software engineering 23, 5 (1997), 279–295.
[19] Marc Jasper, Malte Maes, Alina Murtov, Maximilian Schlüter, Falk Howar, Bernard Stoffen, Markus Schordon, Dennis Hendriks, Ramon Schüffers, Marco Kuppens, et al. 2019. BERS 2019: combining synthesis with real-world models. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 101–115.
[20] Jurie Leskovec, Kiyu Yu, Weihua Hu and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks?. In International Conference on Learning Representa- tions.
[21] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimi- zation. In International Conference on Learning Representations.
[22] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In International Conference on Learning Representations.
[23] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kuhitij Bansal, and Christian Szegedy. 2020. Graph Representations for Higher-Order Logic and Theorem Proving. In The Thirty-Fourth AAAI Conference on Artificial Intelligence. AAAI Press, 2967–2974.
[24] Carl Adam Petri. 1983. Some Personal Views of Net Theory. In Applications and Theory of Petri Nets. Springer, 1–13.
[25] Amir Pnueli. 1977. The Temporal Logic of Programs. In Proceedings of the 18th International Conference on Automated Software Engineering. 495–499.
[26] Ulsha Ruby and VamsiDhar Varendapalli. 2020. Binary cross entropy with deep learning technique for image classification. Int. J. Adv. Trends Comput. Sci. Eng. 9, 10 (2020).
[27] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. 2018. How does batch normalization help optimization? Advances in neural information processing systems 31 (2018).
[28] A Saul. 1963. Kripke. Semantical analysis of modal logic I: Normal modal propositional calculi. Zeitschrift für mathematische Logik und Grundlagen der Mathematik 9, 5-6 (1963), 67–96.
[29] Daniel Selman, Matthew Lamm, Benedict Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. 2019. Learning a SAT Solver from Single-Bit Supervision. arXiv:1802.03685 [cs.AI].
[30] Yassun Tsukamato, Aditya Kanade, Rahul Kumar, Akash Lal, and Aditya V. Nori. 2014. MUX: algorithm selection for software model checkers. In Proceedings of the 31th Working Conference on Mining Software Repositories. ACM, 132–141.
[31] Annti Valmari. 1992. A stubborn attack on state explosion. Formal Methods in System Design 1, 4 (1992), 285–322.
[32] Weijun Zhu, Huanmei Wu, and Miaolei Deng. 2019. LTL Model Checking Based on Binary Classification of Machine Learning. IEEE Access 7, 2019, 135703–135719.
A TRADITIONAL MODEL CHECKING AND TEMPORAL OPERATORS

A.1 Büchi Automata

In automata theory, a Büchi automaton (BA) is a system that either accepts or rejects inputs of infinite length. The automaton is represented by a set of states (one 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 alphabets that hold in the transition. The system 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 systems in this paper, as they are strictly more powerful than deterministic Büchi automata systems.

A non-deterministic Büchi automaton $B$ is formally defined as the tuple $(Q, \Sigma, A, q_0, q_f)$, where $Q$ is the set of all states of $B$, and is finite; $\Sigma$ is the finite set of alphabets; $A : Q \times 2^\Sigma \rightarrow 2^Q$ is the transition relation that can map a state to a set of states on the same input set; $q_0 \in Q$ represents the initial state; $q_f \subset Q$ is the set of final states.

A.2 Linear Temporal Logic

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 (!), and ($\&$), and or ($|$), true (1), false ($\bot$), and temporal operators. Additional logical operators such as implies, equivalence, etc. that can be replaced by the combination of basic logical operators ($\downarrow, \&$, $|$).

For example, $a \rightarrow b$ is expressed as $!a \mid b$.

A.3 LTL Model Checking

Given a Büchi automaton $B$ (system), and an LTL formula $\phi$ (specification), the model checking problem decides whether $B$ satisfies $\phi$. Traditionally, the problem is solved by computing the Büchi automaton for the negation of the specification $\neg \phi$ as $B_{\neg \phi}$, followed by the product automaton $B' = B \times B_{\neg \phi}$. The problem then reduces to checking the language emptiness of $B'$. The language accepted by $B'$ is said to be empty if and only if $B'$ rejects all inputs.

Construction of $B$ is linear in the size of its state space, while $B_{\neg \phi}$ is exponential in the size of $\neg \phi$. The product construction would also lead to an automaton of size $|B| \times 2^{|\neg \phi|}$, which can blow up even $\phi$ is linear in the size of $B$, leading to the state space explosion problem.

Figure 1(a) represents the system $B$ with $\phi$: ‘a U !b’, and the cross product $B' = B \times B_{\neg \phi}$ is given in Figure 1(c). $B'$ does not accept anything as there is no feasible path from the initial state $(q_0' f_0)$ to either of the final states $(q_f' f_0, q_f' f')$. Here, both $B$ and $B_{\neg \phi}$ are three times smaller than $B'$ in terms of the number of states, and 6 times smaller regarding the number of transitions. As observed in Figure 1(c), it can be concluded that even for moderately complex specifications, the product can still result in an exponential state space, which would severely hinder the performance of traditional model checkers.

The meaning of temporal operators supported by $\phi$ is presented in Table 7.

Table 7: Temporal Operators in Linear Temporal Logic

| Symbol | G   | F   | R   | W   | M   | X   | U   |
|--------|-----|-----|-----|-----|-----|-----|-----|
| Meaning | globally | finally | release | weak until | strong release | next until |

B EXPERIMENTAL SETTINGS AND DATASETS

B.1 Environment

Experiments were performed on a cluster with four Intel 24-Core Gold 6248R CPUs, 1TB DRAM, and eight NVIDIA QUADRO RTX 6000 (24GB) GPUs.

B.2 Implementation Details

The code base is implemented on PyTorch 1.8.0 and pytorch-geometric [13]. The source files are attached along with the submitted supplementary.

B.3 Datasets Statistics of Synth and RERS

The specifications for Synth are synthetically generated through Spot [12], while RERS is obtained from the annual competition RERS19 [19]. The data generated from Spot and provided by RERS19 are specifications, i.e., a set of LTL formulae $\phi$’s. To generate the corresponding automaton $B$ for each $\phi$, the tool LTL3BA [2] is used, due to its superior speed [5]. Synth consists of specifications, all of which can be solved by LTL3BA within the time limit (2 mins). Synth aims to test the checking speed of models w.r.t LTL3BA, along with generalization ability for varied length of specifications. RERS is a standard benchmark for sequential LTL from the RERS19 competition. The datasets are catered corresponding to the two scenarios we evaluate OCTAL on. The details of the construction of yes/no pairs for Synth and RERS corresponding to the two applications are discussed in detail in the following sections.

B.3.1 Datasets SynthGen and RERSGen. These datasets correspond to the general model checking scenario, where the yes instances correspond to the system being equivalent to the specification, and the system strictly implying the specification (Figures 2(b), 2(a)). Hence, every specification $\phi$ is paired with systems $B_1$, $B_2$, $B_3$ and $B_4$, where $B_1$ is equivalent to $\phi$, $B_2$ implies but is not equivalent to $\phi$, $B_3$ and $B_4$ do not imply $\phi$ (Figures 2(c), 2(d), 2(e)). Hence, $(B_1, \phi)$ and $(B_2, \phi)$ have label 1, whereas $(B_3, \phi)$ and $(B_4, \phi)$ have label 0. For SynthGen, $\phi$ and $B$ correspond to Synth, and for RERSGen $\phi$ and $B$ correspond to RERS.

B.3.2 Datasets SynthSpec and RERSSpec. These datasets correspond to the special model checking scenario, where we evaluate OCTAL on a strict subset of model checking, where the system satisfies the specification, iff they are equivalent. Hence, every $\phi$ is
paired with systems $B_1$ and $B_2$, where $B_1$ is equivalent to $\phi$ and $B_2$ is not. Hence, $(B_1, \phi)$ has label 1 and $(B_2, \phi)$ has label 0. We consider an equal distribution of strict implication (2(a)) and other and non-implication (2(c), 2(d), 2(e)) tuples for the 0 cases. For SynthSpec, $\phi$ and $B$ correspond to Synth, and for RERSSpec $\phi$ and $B$ correspond to RERS.

### B.3.3 Synth generation details

The `randltl` feature of Spot controls the length of generated LTL formulae, where the default size of expression tree is set to 15. The output formulae are not syntactically the same. The specifications generated through Spot have LTL formulae of lengths (noted as #Lens) ranging from 1 to 80. The length distribution of the formula for Synth is plotted in Figures 4(a). The number of states and edges are described in Figures 4(b), 4(c), respectively. By observing those distributions, it can be concluded that the range of length and states of LTL formulae of RERS is subsumed by Synth. The corresponding transition range is less than 160 for RERS while 1,711 for Synth.

### B.3.4 RERS generation details

Rigorous Examination of Reactive Systems (RERS) is an international model checking competition track organized every year. We adopt 900 specifications from the Sequential LTL track, generate the corresponding BAs and construct the dataset for the two cases. The statistical details of RERS is presented in Figure 5.

### C FURTHER EXPERIMENTS AND ANALYSIS

#### C.1 Generalization on SynthGen

In this experimental setting, we evaluate the performance of out of distribution data. From the distributions of Synth and RERS in section B.3, we observe that Synth strictly subsumes RERS, hence we test how well OCTAL performs inference on SynthGen when trained with RERSGen. From the results in Table 8 we observe that the values of accuracy, precision and recall are similar to the scenario in Table 3 where both training and inference is done on SynthGen. Hence, we can conclude that it generalizes similarly to out of distribution data, as it does for data points in the same distribution.

#### C.2 Analysis of Precision and Recall

To better understand the results of SynthGen wrt the difference between Precision and Recall, we plotted the correct (Figure 6(b)) and incorrect (Figure 6(a)) predictions for the inference of SynthGen. We can observe that the true labels have been wrongly classified more than the false labels, thus yielding more false negatives which results in a low value of Recall. We further investigated the false negatives and found that all of the miss classified true labels belong to the strict implication case, i.e., the scenario in which the $B$ strictly implies $\phi$ (Figure 2(a)). This leads us to conclude that OCTAL performs and generalizes very well when the distribution of the test set is subsumed by the train set (RERSGen), but has some difficulty in classifying the implications correctly otherwise.
Figure 4: Statistical Summary of Synth.
Figure 5: Statistical Summary of RERS.

Figure 6: Figures 6(a) and 6(b) represent the correct and incorrect predictions for SynthGen. The x-axis represents the number of nodes in $G$ and the y-axis represents the number of misclassified labels. The labels correspond to true labels, hence represent False Positives and False Negatives for 6(a), and True Positives and True Negatives for 6(b).