# FoMo - Formula and Model Generation for Learning-Based Formal Methods

Colin Shea-Blymyer
Houssam Abbas
sheablyc@oregonstate.edu
houssam.abbas@oregonstate.edu
Oregon State University
Corvallis, Oregon, USA

## Abstract

This paper presents a tool that can generate system model and temporal logic formula pairs such that the formula is satisfied or unsatisfied as desired. Large amounts of independent and identically distributed data is needed for the development of machine learning approaches in formal methods. However, existing data sets of system models and specifications were hand-designed to challenge existing algorithms for formal methods, and so these data sets represent only a small subset of problems. Learning-based approaches to formal methods require training data drawn from a broader distribution. In this work we introduce a tool called "FoMo" (for "Formula and Model") that generates system models from graph distributions and can sample a specification language to generate properties. FoMo includes functions to generate pairs of formulas and satisfying or unsatisfying system models, and to generate traces from systems. The tool has features for working with linear temporal logic, and with weighted automata. We demonstrate the use of this tool by training a neural network to jointly embed formulas and models to allow for model checking as a classification task. The capabilities offered by FoMo allow researchers to generate system models and properties according to many research needs, and helps provide machine learning systems with generated, multi-modal data sets.

***CCS Concepts:*** • **Computing methodologies** → **Neural networks**; • **Theory of computation** → **Verification by model checking**; Logic and verification; *Modal and temporal logics.*

***Keywords:*** datasets, formal methods, neurosymbolic reasoning

## 1 Introduction

As cyber-physical systems become larger, and more complex, the need to verify their safety and security has not diminished. However, many important formal methods for verification struggle to scale well with the increasing size of these systems. This crucial issue has seeded interest in machine learning (ML) based methods for verification.

Earlier works that aimed to scale formal methods with statistical methods relied on making statistical claims based on samples drawn from a problem [5]. Modern ML-based methods have used recurrent neural networks [7], graph neural networks [10], and transformers [6, 12] to learn to solve problems in formal methods. In [7] and [12], neural networks are used to improve control tasks. A transformer is trained to produce a satisfying trace given an LTL formula in [6], and [10] presents a graph neural network to perform model checking.

Each of these previous works used a different sets of tools to generate large sets of data. This means that new research in this area would have to cobble together new sets of tools to create training data.

We introduce FoMo (short for "Formula and Model") as a first step towards solving these data requirements with a single tool, and demonstrate its use by training a neural network to perform LTL model checking. FoMo is a tool written in python with three main components that generate:

- *System Models* from a graph distribution and set of atomic propositions. Optionally, they may be weighted, or made to satisfy a given LTL formula.
- *Formulas* in LTL from a given set of atomic propositions that are made to be satisfied or to be unsatisfied by a given system model.
- *Traces* from a system model.

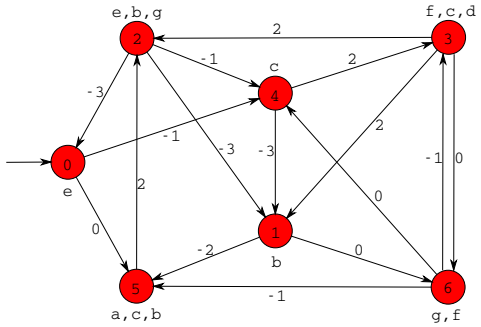
These components can be used in concert to generate data sets of formulas and traces that satisfy them, pairs of formulas and system models labeled with the satisfaction relation between them, system models and lists of formulas they satisfy, etc. The code for this tool is available at https://github.com/sabotagelab/FoMo.

## 2 Inputs and Outputs of FoMo

FoMo has three primary output types: formulas, systems, and traces. Formulas and traces are represented as strings, and systems are `Automaton` objects. A function exists to generate each of these outputs. Further, two functions exist to generate sets of formulas and systems for certain ML training tasks.

### 2.1 Generating Models

The `generate_automaton` function takes a desired number of states, a probability that a transition exists between any




**(a)** An example system model generated by `generate_automaton`.

$$(!f)\ \mathsf{U}\ c$$

**(b)** An example formula generated by `generate_formula`.

**Figure 1.** Example outputs from FoMo

two states, a list of strings that defines the atomic propositions that can label a state, and the maximum number of labels a state can have. Optionally, the function can also take a string representing an LTL formula and a Boolean value that defines a satisfaction relationship between the formula and the model. This function returns an `Automaton` object.

FoMo generates a graph from the Erdős-Rényi distribution designated by this function's first two arguments using igraph [2]. Then the function labels each state with a random number of randomly selected atomic propositions (between 1 and the designated maximum). If a formula was given, then FoMo uses rejection sampling to select a model that has the designated satisfaction relationship with that formula.

***Model Generation Example.*** The weighted automaton in Figure 1a is the result of calling `generate_automaton` for 7 vertices, an edge probability of 0.3, AP symbols 'a' through 'g', between 1 and 3 symbols per state, and edge weights as integers from $[-3, 3)$.

### 2.2 Generating Formulas

The `generate_formula` function takes an `Automaton` object, a list of strings that defines the atomic propositions available to a formula, a maximum formula length, and a Boolean value that defines the desired satisfaction relationship between the `Automaton` object and the generated formula. This function returns a string representing a random LTL formula that respects the given inputs.

FoMo uses SPOT [4] to generate a formula with the given atomic propositions and maximum length. The function then performs rejection sampling to ensure the formula has an appropriate length and the specified satisfaction relationship. Model checking is done with nuXmv [1].

***Formula Generation Example.*** The formula in Figure 1b is the result of calling `generate_formula` for the automaton in Figure 1a, its same set of atomic propositions, and a maximum formula parse tree size of 7.

### 2.3 Generating System Traces

The `generate_trace` function takes an `Automaton` object, the required length for the trace, and a discount factor for the discounted sum of weights accumulated along the trace. The function returns a list of transitions taken, the discounted sum of weights on those transitions, and a list of atomic propositions as they were encountered by the random walk.

### 2.4 Generating Training Data

The `generate_mfl_entry` function and `generate_contrastive_mfl_entry` function both take a list of strings that defines the atomic propositions available to formulas and systems, a desired number of states for systems, a probability that a transition exists between any two states, the maximum number of labels a state can have, and a maximum formula length. Additionally, `generate_contrastive_mfl_entry` takes a number to specify how many systems should be generated that do not satisfy the generated formula. `generate_mfl_entry` generates a system model as a matrix representation, a formula, and a Boolean value indicating if the model satisfies the formula. `generate_contrastive_mfl_entry` generates a system model matrix, a formula that the model satisfies, and a given number of system models that do not satisfy the formula.

It takes about a minute to produce 10,000 contrastive data entries with formulas and systems in a similar form as the examples shown so far.

## 3 Data Distribution

An example use case for FoMo is training an ML algorithm to perform model checking. The algorithm would take two inputs (a system model $\mathcal{M}$, and a formula $\phi$), and output 1 if $\mathcal{M} \models \phi$, and 0 otherwise. To train this algorithm, many thousands of formulas, models, and their satisfaction relationship would have to be generated. To demonstrate the distribution of formulas generated by FoMo, we generated $2^{16}$ pairs of system models and formulas using the `generate_mfl_entry` fucntion. Half of the formulas were selected at random to be satisfied by its paired system model, while the other half were selected to be unsatisfied. This allows us to produce a balanced data set with some interesting statistical properties. Models were sampled from the Erdős-Rényi distribution, with 20 states, a connection probability of 0.3, 11 atomic propositions, and a maximum of 11 labels per state.

The summary statistics (Table 1) show that satisfied formulas and unsatisfied formulas exhibit similar numbers of production rules, terminals, operators, and atomic propositions. Figure 2 provides further detail on the number of

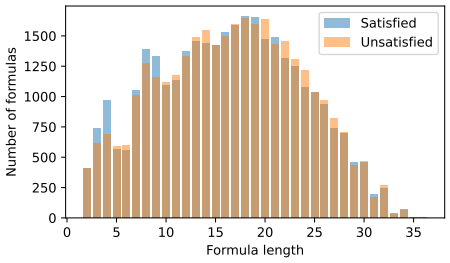



**Figure 2.** Overlapping histograms of formula lengths, measured by the number of terminal symbols. Satisfied formulas are plotted in blue, and unsatisfied formulas in orange. Bars with blue tops (as in the bar for length 4) indicate that more satisfied formulas had that number of terminals; and more unsatisfied formulas did if orange. In the case of formulas for length 4, this chart shows that fewer than 750 unsatisfied formulas had exactly 4 terminals, but almost 1000 satisfied formulas did.

**Table 1.** Average number (with standard deviation) of production rules, terminal symbols, operators, and atomic propositions per each formula generated.

| | all formulas | satisfied formulas | unsatisfied formulas |
|---|---|---|---|
| production rules | 26.90 (13.11) | 26.62 (13.12) | 27.18 (13.10) |
| terminals | 16.66 (7.25) | 16.30 (7.30) | 16.59 (7.19) |
| operators | 7.12 (3.08) | 6.86 (3.07) | 7.38 (3.07) |
| atomic propositions | 4.28 (1.61) | 4.41 (1.62) | 4.15 (1.59) |

terminals among satisfied and unsatisfied formulas. We can see, for example, that smaller formulas are more likely to be satisfied on this distribution of system models.

## 4 Neural Model Checking

Using the data generated by the `generate_contrastive_mfl_entry`, we train a transformer model [11] to embed models and formulas in the same latent space, and a head to perform the classification task of model checking. The embedding network is a BERT encoder model [3] trained with contrastive loss [8]. We used 4 hidden layers and 4 attention heads, an embedding size (hidden size) of 128, and a learning rate of $3 \times 10^{-3}$. The training data consists of 131072 entries, each composed of one LTL formula, one system that satisfies the formula, and 5 systems that do not satisfy the formula. The data has 4 atomic propositions, the formulas are length 7, and the systems have 10 states, and up to 4 atomic propositions on each state. The network was trained for 64 epochs with a batch size of 32 and 32 gradient accumulation steps. The classification head is composed of two fully connected layers connected by ReLU activations, and was trained with cross entropy loss independently of the embedding network for 10 epochs. We evaluated the network with graphs from the Barabasi distribution to avoid the inclusion of training examples in the evaluation set.

The encoder model achieved a training loss of 1.04, and an evaluation loss of 1.31. The classification head achieved an accuracy of 69% at a loss of 0.63. Though the embedding network seemed to struggle with generalizing, the performance of the classification head suggests that the embeddings contain some useful information. We expect that further fine-tuning of the embedding network, using larger data sets with more contrastive examples would lead to better generalization of the embedding network and performance in the model checking head.

## 5 Conclusion

FoMo can be used to generate data sets to train and test learning-based formal methods algorithms. By combining function to generate formulas, system models, and traces, users can produce data sets of labeled pairs of formulas and systems, traces that satisfy a given formula, or systems that don't satisfy a given formula.

We have shown that such data can be used to train a model that jointly embeds formulas and models. Such networks could be used to perform tasks such as formula search, or model repair by interpolating between embeddings of specifications and systems.

We would like to expand the functionality of FoMo along multiple directions. The addition of support for more specification languages and associated system models would enable a broader class of learning-based algorithms, and could promote better generalization of such algorithms. A better selection of distributions to draw automata from would improve the robustness of algorithms trained on data generated by FoMo, and would allow users to better model their anticipated use-case. Works such as [9] use GANs to sample harder-to-solve specification problems, and we would like to apply similar techniques to the generation of automata as well. The development of sampling methods that go beyond rejection sampling could increase the performance of FoMo, allowing for larger, more complex data sets to be generated.

## References

[1] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv symbolic model checker. In International Conference on Computer Aided Verification. Springer, 334–342.

[2] Gabor Csardi, Tamas Nepusz, et al. 2006. The igraph software package for complex network research. InterJournal, complex systems 1695, 5 (2006), 1–9.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018).

[4] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, et al. 2022. From Spot 2.0 to Spot 2.10: What's New?. In International Conference on Computer Aided Verification. Springer, 174–187.

[5] Radu Grosu and Scott A Smolka. 2005. Monte carlo model checking. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 271–286.

[6] Christopher Hahn, Frederik Schmitt, Jens U Kreber, Markus N Rabe, and Bernd Finkbeiner. 2020. Teaching temporal logics to neural networks. arXiv preprint arXiv:2003.04218 (2020).

[7] Wataru Hashimoto, Kazumune Hashimoto, and Shigemasa Takai. 2022. STL2vec: Signal temporal logic embeddings for control synthesis with recurrent neural networks. IEEE Robotics and Automation Letters 7, 2 (2022), 5246–5253.

[8] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. Advances in neural information processing systems 33 (2020), 18661–18673.

[9] Jens U Kreber and Christopher Hahn. 2021. Generating symbolic reasoning problems with transformer gans. arXiv preprint arXiv:2110.10054 (2021).

[10] Prasita Mukherjee, Haoteng Yin, Susheel Suresh, and Tiark Rompf. 2022. OCTAL: Graph Representation Learning for LTL Model Checking. arXiv preprint arXiv:2207.11649 (2022).

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. Advances in neural information processing systems 30 (2017).

[12] Yaqi Xie, Fan Zhou, and Harold Soh. 2021. Embedding symbolic temporal knowledge into deep sequential models. In 2021 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 4267–4273.

Received 3 May 2023