Learning to Check LTL Satisfiability and to Generate Traces via Differentiable Trace Checking

Weilin Luo
School of Computer Science and Engineering, Sun Yat-sen University
Guangzhou, China
luowlin5@mail.sysu.edu.cn

Pingjia Liang
School of Computer Science and Engineering, Sun Yat-sen University
Guangzhou, China
liangpj3@mail2.sysu.edu.cn

Junming Qiu
School of Computer Science and Engineering, Sun Yat-sen University
Guangzhou, China
qiujm9@mail2.sysu.edu.cn

Polong Chen
School of Computer Science and Engineering, Sun Yat-sen University
Guangzhou, China
chenplong@mail2.sysu.edu.cn

Hai Wan*
School of Computer Science and Engineering, Sun Yat-sen University
Guangzhou, China
wanhai@mail.sysu.edu.cn

Jianfeng Du*
Guangdong University of Foreign Studies, Guangzhou, China
Bigmath Technology
Shenzhen, China
jfdu@gdufs.edu.cn

Weiyuan Fang
School of Computer Science and Engineering, Sun Yat-sen University
Guangzhou, China
fangwy3@mail2.sysu.edu.cn

Abstract

Linear temporal logic (LTL) satisfiability checking has a high complexity, *i.e.*, PSPACE-complete. Recently, neural networks have been shown to be promising in approximately checking LTL satisfiability in polynomial time. However, there is still a lack of neural network-based approach to the problem of checking LTL satisfiability and generating traces as evidence, simply called *SAT-and-GET*, where a satisfiable trace is generated as evidence if the given LTL formula is detected to be satisfiable. In this paper, we tackle SAT-and-GET via bridging LTL trace checking to neural network inference. Our key theoretical contribution is to show that a well-designed neural inference process, named after *neural trace checking*, is able to simulate LTL trace checking. We present a neural network-based approach `VSCNet`. Relying on the differentiable neural trace checking, `VSCNet` is able to learn both to check satisfiability and to generate traces via gradient descent. Experimental results confirm the effectiveness of `VSCNet`, showing that it significantly outperforms the state-of-the-art (SOTA) neural network-based approaches for trace generation, on average achieving up to 41.68% improvement in semantic accuracy. Besides, compared with the SOTA logic-based approach `nuXmv` and `Aalta`, `VSCNet` achieves averagely 186X and 3541X speedups on large-scale datasets, respectively.

*Both authors are corresponding authors.

CCS Concepts

• **Theory of computation → Modal and temporal logics**; • **Computing methodologies → Neural networks**.

Keywords

linear temporal logic, satisfiability checking, neural networks, trace checking

ACM Reference Format:
Weilin Luo, Pingjia Liang, Junming Qiu, Polong Chen, Hai Wan, Jianfeng Du, and Weiyuan Fang. 2024. Learning to Check LTL Satisfiability and to Generate Traces via Differentiable Trace Checking . In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680337

1 Introduction

Linear temporal logic (LTL) satisfiability checking aims to check whether a given LTL formula is satisfiable. It is a fundamental theoretical problem about LTL and has important applications in formal verification, *e.g.*, model checking [16]. For example, it is used to check requirements sanity [4], *i.e.*, to ensure a given set of requirements to be consistent. Besides, it has widely been applied in software engineering, *e.g.*, goal-conflict analysis [15, 35] and business process [38]. However, checking LTL satisfiability is PSPACE-complete [54], which means that, it will inevitably suffer from the search space explosion problem.

A variety of logic-based approaches have been proposed for checking LTL satisfiability, *e.g.*, based on model checking [44, 45], tableau [5, 24, 51, 61], temporal resolution [19], anti-chains [62], and Boolean satisfiability problem (SAT) [26–30]. The performance of these logic-based approaches heavily relies on well-design search heuristics to alleviate the exploding search space. However, it is hard to design good search heuristics and to verify their ubiquitous effectiveness. Even worse, given the inherent intractability of checking LTL satisfiability, there is no search heuristics that can ensure the checking to be done efficiently in all cases.

Inspired by the superior learning ability of neural networks, recent research on LTL satisfiability checking has migrated from hand-engineered approaches toward data-driven ones. Luo et al. [36] have shown that neural networks are promising approximators for checking LTL satisfiability in polynomial time. Further, they [37] studied the problem of checking LTL satisfiability and generating traces as evidence, simply called *SAT-and-GET*, where if a satisfiable outcome is obtained, an additional trace will be generated to logically check against the formula to confirm the satisfiability. The generation of satisfiable traces is crucial for some domains, *e.g.*, safety-critical systems, counterexample-driven approaches, and incremental LTL satisfiability checking. In order to generate satisfiable traces, they [37] proposed an approach based on deep reinforcement learning (RL), where an end-to-end neural network is trained to generate satisfiable traces, relying on feedback from the environment implemented as a polynomial-time trace checking algorithm [39]. It has been pointed out [1, 3, 23, 59] that RL is usually sample inefficient and suffers from the notorious problem of unstable training, due to the feedback of the environment being incompatible with the gradient descent-based learning mechanism in neural networks. As a result, the proposed approach in [37] can only achieve slight performance improvements beyond random guessing for trace generation. Therefore, by now there is still a lack of study on tractable and more effective approaches to the SAT-and-GET problem.

It is challenging to teach neural networks to generate a satisfiable trace from a given formula. One of the challenges lies in bridging the gap between neural networks defined in the continuous domain and the trace representation formalized in the discrete domain. This gap makes neural networks hard to capture features for logic inference [33, 60]. Another challenge lies in the fact that there can be multiple satisfiable traces for a single satisfiable LTL formula. It is unclear which particular satisfiable traces should be supervised to drive the training of trace generation. Therefore, supervised training of a neural model for generating a satisfiable trace tends to introduce negative bias and thus may not be reasonable.

We tackle these challenges by bridging LTL trace checking to neural network inference. We develop an LTL encoding method to parameterize a neural network and theoretically prove that its inference process, named after *neural trace checking*, is able to simulate LTL trace checking. Neural trace checking models the checking of the satisfaction relation between a trace and an LTL formula. It enables a novel approach, VSCNet, to solving the SAT-and-GET problem, which jointly trains a neural network for LTL satisfiability checking (SCNet) and a neural network for trace generation (TGNet). The key idea of TGNet is to use the implicit input of neural trace checking as weak supervision, thus avoiding the selection of specific satisfiable traces as supervision in training TGNet.

Following the evaluation setups in [36], we conduct extensive experiments on both synthetic datasets and large-scale datasets.[1] Experimental results demonstrate that our approach is much more efficient than the state-of-the-art (SOTA) logic-based approaches while achieving comparable performance; besides, it also achieves significantly higher performance than SOTA neural network-based approaches in generating satisfiable traces.

In general, our contributions can be summarized as follows.

- We propose an LTL encoding method to parameterize a neural network and theoretically prove that its inference process, *i.e.*, neural trace checking, is able to simulate LTL trace checking.
- We propose a novel approach, VSCNet, to jointly train a neural network for satisfiability checking and a neural network for trace generation.
- Experimental results confirm the effectiveness of VSCNet, which spends much shorter time to achieve comparable performance with SOTA logic-based approaches in LTL satisfiability checking and achieves higher performance than SOTA neural network-based approaches for trace generation.

2 Background

2.1 Linear Temporal Logic

The syntax of *Linear temporal logic (LTL)* [42] over a finite set of atomic propositions $\mathbb{P}$ is defined by logical operators $\wedge$ and $\neg$ as well as temporal modal operators next ($\bigcirc$) and until ($\mathcal{U}$):

$$\phi := p \mid \neg\phi' \mid \phi' \wedge \phi'' \mid \bigcirc\phi' \mid \phi' \,\mathcal{U}\, \phi'',$$

where $p \in \mathbb{P} \cup \{\top\}$, and ϕ' and ϕ'' are LTL formulae. For brevity, we only consider these fundamental operators in this paper. Operator *or* ($\vee$), *release* ($\mathcal{R}$), *eventually* ($\Diamond$), *always* ($\Box$), and *weak-until* ($\mathcal{W}$) can be defined as $\phi' \vee \phi'' := \neg(\neg\phi' \wedge \neg\phi'')$, $\phi' \,\mathcal{R}\, \phi'' := \neg(\neg\phi' \,\mathcal{U}\, \neg\phi'')$, $\Diamond\phi' := \top \,\mathcal{U}\, \phi'$, $\Box\phi' := \neg(\top \,\mathcal{U}\, \neg\phi')$, and $\phi' \,\mathcal{W}\, \phi'' := (\phi' \,\mathcal{U}\, \phi'') \vee \Box\phi'$, respectively. Moreover, $\top$ can be defined as $p \vee \neg p$ for any $p \in \mathbb{P}$. Hence, although we only consider the fundamental operators throughout the paper, our approach can indeed accept all other temporal modal operators.

Let ϕ be an LTL formula. The *size* of ϕ, denoted by $|\phi|$, is recursively defined as follows: if $\phi = p$, then $|\phi| = 1$; if $\phi = \circ_1\, \phi'$, then $|\phi| = |\phi'| + 1$; if $\phi = \phi' \circ_2 \phi''$, then $|\phi| = |\phi'| + |\phi''| + 1$, where $p \in \mathbb{P}$, $\circ_1 \in \{\neg, \bigcirc\}$, $\circ_2 \in \{\wedge, \mathcal{U}\}$, and ϕ', ϕ'' are LTL formulae. The set of *sub-formulae* of ϕ, denoted by $\mathtt{sub}(\phi)$, is recursively defined as follows: if $\phi = p$, then $\mathtt{sub}(\phi) = \{p\}$; if $\phi = \circ_1\, \phi'$, then $\mathtt{sub}(\phi) = \{\phi\} \cup \mathtt{sub}(\phi')$; if $\phi = \phi' \circ_2 \phi''$, then $\mathtt{sub}(\phi) = \{\phi\} \cup \mathtt{sub}(\phi') \cup \mathtt{sub}(\phi'')$, where $p \in \mathbb{P}$, $\circ_1 \in \{\neg, \bigcirc\}$, $\circ_2 \in \{\wedge, \mathcal{U}\}$, ϕ', ϕ'' are LTL formulae.

The *syntax tree* of ϕ is a 5-tuple $(V, E, v_1, \mathtt{lab}_V, \mathtt{lab}_E)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of undirected edges, $v_1 \in V$ is the root vertex, $\mathtt{lab}_V : V \longrightarrow \mathbb{P} \cup \{\neg, \wedge, \bigcirc, \mathcal{U}\}$ is a labeled function of vertices and $\mathtt{lab}_E : E \longrightarrow \{L, R\}$ is a labeled function of edges, and where L and R denote the left child and the right child, respectively. V and E are initialized as $\{v_1\}$ and $\emptyset$, respectively. For every $\phi_i \in \mathtt{sub}(\phi)$, where $i \in [1, |\phi|]$ and $\phi_1 = \phi$, $(V, E, v_1, \mathtt{lab}_V, \mathtt{lab}_E)$ is constructed as follows: if $\phi_i = p$, then $\mathtt{lab}_V(v_i) = p$ for v_i a leaf vertex; if $\phi_i = \circ_1\phi_j$, then $V = V \cup \{v_j\}$, $E = E \cup \{(v_i, v_j)\}$, $\mathtt{lab}_V(v_i) = \circ_1$, and $\mathtt{lab}_E((v_i, v_j)) = L$; if $\phi_i = \phi_j \circ_2 \phi_k$, then $V = V \cup \{v_j, v_k\}$, $E = E \cup \{(v_i, v_j), (v_i, v_k)\}$, $\mathtt{lab}_V(v_i) = \circ_2$, $\mathtt{lab}_E((v_i, v_j)) = L$ and $\mathtt{lab}_E((v_i, v_k)) = R$, where $p \in \mathbb{P}$, $\circ_1 \in \{\neg, \bigcirc\}$, $\circ_2 \in \{\wedge, \mathcal{U}\}$, and ϕ_j, ϕ_k are LTL formulae. By $\mathtt{tree}(\phi)$ we denote the syntax tree of

[1]Code and data are available at: https://github.com/sysulic/VSCNet.

ϕ. For example, given $\phi = p_1 \, \mathcal{U} \, \bigcirc p_2$, $\mathtt{tree}(\phi)$ is as follows: $V = \{v_1, v_2, v_3, v_4\}$; $E = \{(v_1, v_2), (v_1, v_3), (v_3, v_4)\}$; $\mathtt{lab}_V(v_1) = \mathcal{U}$, $\mathtt{lab}_V(v_2) = p_1$, $\mathtt{lab}_V(v_3) = \bigcirc$, and $\mathtt{lab}_V(v_4) = p_2$; $\mathtt{lab}_E((v_1, v_2)) = L$, $\mathtt{lab}_E((v_1, v_3)) = R$, and $\mathtt{lab}_E((v_3, v_4)) = L$.

Given an LTL formula ϕ, by $\mathtt{pretravel}(\mathtt{tree}(\phi))$ we further denote the sequence of vertices of $\mathtt{tree}(\phi)$ by preorder traversal where sub-trees are travelled from left to right. For the aforementioned example $\phi = p_1 \, \mathcal{U} \, \bigcirc p_2$, $\mathtt{pretravel}(\mathtt{tree}(\phi)) = \langle v_1, v_2, v_3, v_4 \rangle$.

LTL formulae are interpreted over infinite traces. A *trace* is represented of the form $\pi = \pi[1], \ldots, (\pi[k], \ldots, \pi[n])^\omega$, where $\pi[i] \subseteq \mathbb{P}$ is a *state* at time i, $(\pi[k], \ldots, \pi[n])^\omega$ denotes an infinite *loop* where $\pi[k], \ldots, \pi[n]$ appear in order, and k is the *loop-start time.* n is said to be the *trace size* of π, denoted by $|\pi|$. For every $p \in \mathbb{P}$, p is said to hold in $\pi[i]$ if $p \in \pi[i]$, otherwise $\neg p$ is said to hold in $\pi[i]$. π_t denotes the *sub-trace* of π beginning from the state $\pi[t]$; thus, π_1 also denotes π. The *satisfaction relation* $\models$ is defined as follows:

$$\begin{array}{lll} \pi_t \models p & \text{iff} & p \in \pi[t], \\ \pi_t \models \neg\phi' & \text{iff} & \pi_t \not\models \phi', \\ \pi_t \models \phi' \wedge \phi'' & \text{iff} & \pi_t \models \phi' \text{ and } \pi_t \models \phi'', \\ \pi_t \models \bigcirc\phi' & \text{iff} & \pi_{t+1} \models \phi', \\ \pi_t \models \phi' \, \mathcal{U} \, \phi'' & \text{iff} & \exists k \geq t \ s.t.\ \pi_k \models \phi'' \text{ and} \\ & & \forall t \leq j < k, \pi_j \models \phi', \end{array}$$

where π is a trace, $t \in \mathbb{N}$, ϕ', ϕ'' are LTL formulae, and $p \in \mathbb{P}$. An LTL formula ϕ is *satisfiable* if and only if there is a trace π such that $\pi \models \phi$. Checking LTL satisfiability is to check whether an LTL formula is satisfiable or not, which is PSPACE-complete [54]. LTL trace checking is to decide whether a trace satisfies an LTL formula, which is computable in polynomial time [39].

2.2 Neural Network

Throughout this paper we use lowercase bold letters (*e.g.*, $\boldsymbol{x}$) to represent vectors and uppercase bold letters (*e.g.*, $\boldsymbol{X}$) to represent matrices. $(\boldsymbol{x})_i$ denotes the i^{th} element of a vector $\boldsymbol{x}$ and $(\boldsymbol{X})_{i,j}$ denote the element of a matrix $\boldsymbol{X}$ at the i^{th} row and the j^{th} column, where i and j start from 1.

TreeNN [55, 56] is a recursive neural network, widely used as a general extractor for tree structure. TreeNN learns the feature embedding for each vertex in a given tree by recursively combining the representations of its subtrees. Algorithm 1 shows the framework of TreeNN, where τ_v represents the type of vertex v. The combination function (COMBINE) yields different embeddings according to different types of the current vertex.

Algorithm 1: EXTRACTOR-T

```
    Input: The current vertex $v$.
    Output: The embedding $\mathbf{r}_v$ of $v$.
1 if $v$ is leaf then
2     $\mathbf{r}_v$ ←EMBEDDING($v$)
3 else
4     get the set of child vertices $\{c_1, \ldots, c_k\}$ of $v$
5     $\mathbf{r}$ ←AGGREGATE(EXTRACTOR-T ($c_1$),. . . ,EXTRACTOR-T ($c_k$))
6     $\mathbf{r}_v$ ←COMBINE($\mathbf{r}$,$\tau_v$)
7 return $\mathbf{r}_v$
```

Based on TreeNN, Luo et al. [36] proposed several neural network-based approaches, *i.e.*, `TreeNN-MP`, `TreeNN-con` and `TreeNN-inv`, to check LTL satisfiability. `TreeNN-MP` employs mean pooling to aggregate the embeddings of the sub-formulae. `TreeNN-con` implements the aggregating function by concatenating the embeddings of the sub-formulae in order. `TreeNN-inv` employs the same aggregating function as `TreeNN-con`, but adding a new loss function to enforce exchangeable aggregation results for operators in accord with permutation invariance, *e.g.*, the operator $\wedge$.

Luo et al. [37] proposed *one-step unfolded graph (OSUG)* as a graph representation for LTL formulae. The core idea is to equivalently transform an LTL formula into constraints that need to be satisfied at this time or at the next time and to build a graph-based on the syntax tree of the constraints. Intuitively, such an equivalent representation amounts to a one-step unfolding of the LTL formula. For example, given an LTL formula $p_1 \, \mathcal{U} \, p_2$, its equivalent representation is $p_2 \vee (p_1 \wedge \bigcirc(p_1 \, \mathcal{U} \, p_2))$.

Accordingly, Luo et al. [37] proposed an OSUG-based extractor. It extracts features of OSUG via a sample and aggregate graph convolutional network, *e.g.*, GraphSAGE [22]. Specifically, its input includes an OSUG and initial embeddings of vertices in the OSUG, whereas its output consists of all vertex embeddings and a graph embedding. The embedding of vertex v_i in the t-th iteration, denoted by $\boldsymbol{v}_i^{(t)}$, is computed by

$$\boldsymbol{v}_i^{(t)} = \mathsf{ReLU}(\boldsymbol{W}_\alpha \boldsymbol{v}_i^{(t-1)} + \boldsymbol{W}_\beta \frac{1}{|\mathsf{N}(i)|} \sum_{j \in \mathsf{N}(i)} \boldsymbol{v}_j^{(t-1)}), \qquad (1)$$

where $\boldsymbol{v}_i^{(0)}$ is the initial embedding of v_i, $\boldsymbol{W}_\alpha$ and $\boldsymbol{W}_\beta$ are two learnable weights, ReLU is the ReLU activation function, and $\mathsf{N}(i)$ is the set of neighbors of v_i. The maximum number of iterations is equal to the number of layers of the OSUG-based extractor. Initial embeddings are treated as learnable vectors dependent on the type of vertices. Mean pooling is applied to all vertex embeddings to yield the graph embedding.

3 Related Work

3.1 Logic-Based Approaches

The classic approach to LTL satisfiability checking is based on model checking techniques [44, 45]. Li et al. [30] empirically demonstrated the superiority of the model checker `nuXmv` [10] on checking LTL satisfiability. Although the classic approach can benefit from proven techniques, *e.g.*, bounded model checking [6, 12, 13], interpolation [40], and property directed reachability [9, 17], it may not be optimized in terms of efficiency given that satisfiability checking is merely a special case of model checking.

Another dominant approach to LTL satisfiability checking is based on tableau techniques [5, 51, 61]. Fisher et al. [19] proposed a clausal temporal resolution method for LTL, based on a translation from arbitrary LTL formulae into a normal form preserving satisfiability. Subsequently, the notion of unsatisfiable cores for LTL based on temporal resolution was proposed [47]. Wulf et al. [62] developed a new LTL satisfiability checking algorithm, which works directly with alternating automata by employing efficient exploration techniques based on anti-chains. Li et al. [28] proposed an on-the-fly approach for the LTL satisfiability problem, which is able to find a satisfying model quickly without constructing the full automaton. Li et al. [27] implemented a new LTL satisfiability checker named `Aalta`, leveraging the power of modern SAT solvers. After that, `Aalta` was continuously strengthened. Li et al. [29] and Li et al. [30] developed an improved explicit LTL-satisfiability solver checker, where the temporal transition system is constructed through SAT-solving. Further, Li et al. [26] presented two novel techniques to accelerate LTL satisfiability checking, based on an adaptation of the obligation-set method.

All the above logic-based approaches suffer from the inherent intractability of the LTL satisfiability problem.

3.2 Neural Network-Based Approaches

Neural networks have shown to be promising in application to logical reasoning tasks, such as mathematical reasoning [25, 43], SAT solving [31, 52, 53, 67], and formal language learning [33, 60, 66]. The success of neural networks mainly lies in their ability of capturing heuristics that are hard to define with logical rules.

There exist studies on LTL embedding and reasoning. Vaezipoor et al. [57] encoded LTL instructions via graph neural networks (GNNs) to train an RL agent to learn task-conditioned policies. Xie et al. [64] induced bias on the automata equivalent to LTL formulae to characterize temporal knowledge. Schmitt et al. [46] trained hierarchical Transformers to synthesize hardware circuits in the light of LTL specifications. Cosler et al. [14] proposed a separated hierarchical Transformer to repair defective sequential circuits against given LTL specifications. Mukherjee et al. [41] presented a graph representation learning-based approach for LTL model checking. Although the above studies reveal that neural networks can effectively capture LTL features, there is no evidence that their proposed methods can be applied to solving the SAT-and-GET problem.

There are also studies on exploiting neural networks to predict the satisfiability of LTL formulae [34, 36] and to generate traces [21]. Luo et al. [36] designed neural networks keeping three logical properties of LTL and demonstrated their good generalizability across formula sizes. Luo et al. [34] focused on the satisfiability checking problem of LTL over finite traces (LTL$_f$). They did not consider the problem of generating traces for proving the satisfiability of the formula. Hahn et al. [21] applied a Transformer to generate traces of LTL formulae and empirically demonstrated that their approach can generalize to formulae with larger sizes. This Transformer-based approach, however, has efficiency issue on the token-by-token output and cannot guarantee the correctness of the trace structure. Our experimental results shown in Table 2 expose this weakness. Besides, Luo et al. [37] have attempted to tackle the SAT-and-GET problem. They employed RL to tune the strategy for generating traces. Since RL is not differentiable, such an RL-based approach is time-consuming and hard to converge. In contrast, in this work we design a neural network for the SAT-and-GET problem so that the whole problem can be solved by a single neural network trainable with classical gradient descent. Our experimental results shown in Table 2 demonstrate the advantages of our approach.

4 Neural Trace Checking

The key to a neural network-based approach is bridging the gap between neural network inference and logical inference. To explore a neural network-based approach for the SAT-and-GET problem, we design a special neural network to simulate trace checking. We first propose the notion of LTL encoding that represents an LTL formula, then establish a correspondence between LTL encoding and trace checking.

4.1 LTL Encoding

The notion of LTL encoding is given in Definition 1. Intuitively, the LTL encoding characterizes the syntax tree of an LTL formula. Given an LTL formula ϕ and $\langle v_1, \ldots, v_{|\phi|}\rangle = \texttt{pretravel}(\texttt{tree}(\phi))$, we associate the sub-formula ϕ_i with v_i one by one for $i \in [1, |\phi|]$. Specially, ϕ corresponds to v_1. $(\boldsymbol{\eta}_{right})_{i,j}$ indicates whether the right sub-formula of ϕ_i is ϕ_j; $(\boldsymbol{\eta}_{atom})_{i,j}$ indicates whether ϕ_i is the atomic proposition p_j; $(\boldsymbol{\eta}_{\circ})_i$ indicates whether ϕ_i is defined to connect its sub-formulae via operator $\circ \in \{\neg, \wedge, \bigcirc, \mathcal{U}\}$. The left child of v_i is v_{i+1} in preorder traversal where sub-trees are travelled from left to right. Example 1 illustrates the LTL encoding.

Definition 1. *Let $\mathbb{P}$ be a set of atom propositions, ϕ an LTL formula over $\mathbb{P}$, $(V, E, v_1, \texttt{lab}_V, \texttt{lab}_E) = \texttt{tree}(\phi)$, and $\langle v_1, \ldots, v_{|\phi|}\rangle = \texttt{pretravel}(\texttt{tree}(\phi))$. The LTL encoding η of ϕ is a 6-tuple*

$$\left(\boldsymbol{\eta}_{right}, \boldsymbol{\eta}_{atom}, \boldsymbol{\eta}_{\neg}, \boldsymbol{\eta}_{\wedge}, \boldsymbol{\eta}_{\bigcirc}, \boldsymbol{\eta}_{\mathcal{U}}\right),$$

where $\boldsymbol{\eta}_{right} \in \{0,1\}^{|\phi|\times|\phi|}$, $\boldsymbol{\eta}_{atom} \in \{0,1\}^{|\phi|\times|\mathbb{P}|}$, and $\boldsymbol{\eta}_{\neg}, \boldsymbol{\eta}_{\wedge}, \boldsymbol{\eta}_{\bigcirc}, \boldsymbol{\eta}_{\mathcal{U}} \in \{0,1\}^{|\phi|}$; moreover, for all subscripts i and j, $(\boldsymbol{\eta}_{right})_{i,j} = 1$ if $\texttt{lab}_E((v_i, v_j)) = R$ and $(\boldsymbol{\eta}_{right})_{i,j} = 0$ otherwise, $(\boldsymbol{\eta}_{atom})_{i,j} = 1$ if $\texttt{lab}_V(v_i) = p_j$ and $(\boldsymbol{\eta}_{atom})_{i,j} = 0$ otherwise, $(\boldsymbol{\eta}_{\circ})_i = 1$ if $\texttt{lab}_V(v_i) = \circ$ and $(\boldsymbol{\eta}_{\circ})_i = 0$ otherwise, where $\circ \in \{\neg, \wedge, \bigcirc, \mathcal{U}\}$.

Example 1. *Let $\phi = p_1 \ \mathcal{U} \ \bigcirc p_2$ be an LTL formula. $\texttt{tree}(\phi)$ is as follows: $V = \{v_1, v_2, v_3, v_4\}$; $E = \{(v_1, v_2), (v_1, v_3), (v_3, v_4)\}$; $\texttt{lab}_V(v_1) = \mathcal{U}$, $\texttt{lab}_V(v_2) = p_1$, $\texttt{lab}_V(v_3) = \bigcirc$, and $\texttt{lab}_V(v_4) = p_2$; $\texttt{lab}_E((v_1, v_2)) = L$, $\texttt{lab}_E((v_1, v_3)) = R$, and $\texttt{lab}_E((v_3, v_4)) = L$. $\texttt{pretravel}(\texttt{tree}(\phi)) = \langle v_1, v_2, v_3, v_4\rangle$. The LTL encoding η of ϕ is $(\boldsymbol{\eta}_{\mathcal{U}})_1 = 1$, $(\boldsymbol{\eta}_{atom})_{2,1} = 1$, $(\boldsymbol{\eta}_{\bigcirc})_3 = 1$, $(\boldsymbol{\eta}_{atom})_{4,2} = 1$, $(\boldsymbol{\eta}_{right})_{1,3} = 1$, and all other elements of η are 0.*

4.2 Neural Modeling of Trace Checking

With the LTL encoding, we propose a neural network – `NTCNet`, to model the checking of the satisfaction relation between a trace and an LTL formula. The input to `NTCNet` is a tensor representation of the given trace π, called the *tensorized trace* of π, which is formalized in Definition 2, where $(\boldsymbol{s})_{i,j}$ indicates the probability that the atomic proposition $p_j \in \mathbb{P}$ is true at time i, and $(\boldsymbol{l})_i$ indicates the probability that time i is the loop-start time. Example 2 showcases a tensorized trace.

Definition 2. *Let $\mathbb{P}$ be a set of atom propositions and π a trace over $\mathbb{P}$. The tensorized trace of π is a 2-tuple $(\boldsymbol{s}, \boldsymbol{l})$ computed by Algorithm 2, where $\boldsymbol{s} \in \mathbb{R}_{[0,1]}^{|\pi|\times|\mathbb{P}|}$ and $\boldsymbol{l} \in \mathbb{R}_{[0,1]}^{|\pi|}$, and where $\mathbb{R}_{[0,1]}$ denotes the real value range from 0 to 1.*

Example 2. *Let $\mathbb{P} = \{p_1, p_2\}$ be a set of atom propositions and $\pi = (\{p_1, p_2\}, \{p_1\})^{\omega}$ a trace over $\mathbb{P}$. The tensorized trace $(\boldsymbol{s}, \boldsymbol{l})$ of π is shown as follows:*

$$\boldsymbol{s} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}, \quad \boldsymbol{l} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Algorithm 2: TENSORIZE

```
Input: A trace $\pi$ over a set of atomic propositions $\mathbb{P}$.
Output: The tensorized trace $(\boldsymbol{s}, \boldsymbol{l})$ of $\pi$.
s ← 0 where $\boldsymbol{s} \in \{0,1\}^{|\pi|\times|\mathbb{P}|}$, $\boldsymbol{l} \leftarrow \boldsymbol{0}$ where $\boldsymbol{l} \in \{0,1\}^{|\pi|}$
for each state $\pi[i]$ of $\pi$ do
    for each atomic proposition $p_j \in \mathbb{P}$ do
        if $p_j \in \pi[i]$ then
            $(\boldsymbol{s})_{i,j} \leftarrow 1$
    if $i$ is loop-start time then
        $(\boldsymbol{l})_i \leftarrow 1$
```

NTCNet is exactly parameterized by the LTL encoding of the given LTL formula ϕ in the satisfaction relation to be checked.

Definition 3. *Let π be a trace, ϕ an LTL formula, $(\boldsymbol{s}, \boldsymbol{l})$ the tensorized trace of π, and $\eta = \left(\boldsymbol{\eta}_{right}, \boldsymbol{\eta}_{atom}, \boldsymbol{\eta}_{\neg}, \boldsymbol{\eta}_{\wedge}, \boldsymbol{\eta}_{\bigcirc}, \boldsymbol{\eta}_{\mathcal{U}}\right)$ the LTL encoding of ϕ. NTCNet is parameterized by $\theta = \eta$. The satisfaction vectors $\boldsymbol{x}_i \in \mathbb{R}^{|\phi|}$ where $i \in [1, |\pi|]$ are initialized by $\boldsymbol{0}$. For j decreases from $|\phi|$ to 1, for t increases from 1 to $|\pi|$, and for i increases from 1 to $|\pi|$, $(\boldsymbol{x}_i)_j$ is computed as follows:*

$$\begin{aligned}(\boldsymbol{x}_i)_j = \sigma(\quad & (\boldsymbol{p}_i)_j + \\ & (\boldsymbol{\theta}_{\neg})_j(1-(\boldsymbol{x}_i)_{j+1}) + \\ & (\boldsymbol{\theta}_{\wedge})_j\sigma((\boldsymbol{x}_i)_{j+1}+(\boldsymbol{r}_i)_j-1) + \\ & (\boldsymbol{\theta}_{\bigcirc})_j(\boldsymbol{x}_{i+1})_{j+1} + \\ & (\boldsymbol{\theta}_{\mathcal{U}})_j(\boldsymbol{u}_i)_j^{(|\pi|)} \quad), \\ (\boldsymbol{x}_{|\pi|+1})_{j+1} = \quad & \textstyle\sum_{k=1}^{|\pi|}(\boldsymbol{l})_k(\boldsymbol{x}_k)_{j+1},\end{aligned} \tag{2}$$

where $\sigma(x) = \min(1, \max(0, x))$ is an activation function, and $(\boldsymbol{p}_i)_j$, $(\boldsymbol{r}_i)_j$, and $(\boldsymbol{u}_i)_j^{(t)}$ are computed by Equation (3), Equation (4), and Equation (5), respectively.

$$(\boldsymbol{p}_i)_j = \sum_{k=1}^{|\mathbb{P}|}(\boldsymbol{\theta}_{atom})_{j,k}(\boldsymbol{s})_{i,k}. \tag{3}$$

$$(\boldsymbol{r}_i)_j = \sum_{k=j+2}^{|\pi|}(\boldsymbol{\theta}_{right})_{j,k}(\boldsymbol{x}_i)_k. \tag{4}$$

$$\begin{aligned}(\boldsymbol{u}_i)_j^{(t)} &= \begin{cases}(\boldsymbol{r}_i)_j, & t=1,\\ \sigma(\sigma((\boldsymbol{x}_i)_{j+1}+(\boldsymbol{u}_{i+1})_j^{(t-1)}-1)+ & \textit{otherwise},\\ (\boldsymbol{u}_i)_j^{(1)}), & \end{cases}\\ (\boldsymbol{u}_{|\pi|+1})_j^{(t)} &= \textstyle\sum_{k=1}^{|\pi|}(\boldsymbol{l})_k(\boldsymbol{u}_k)_j^{(t)}.\end{aligned} \tag{5}$$

By NTCNet$((\boldsymbol{s}, \boldsymbol{l}) \,|\, \eta) = (\boldsymbol{x}_1)_1$ we denote the result of neural trace checking of the satisfaction relation between π and ϕ.

Table 1: Converting logical operations to soften forms.

formal logic	$a \wedge b$	$a \vee b$	$\neg a$
soften logic	$\sigma(a+b-1)$	$\sigma(a+b)$	$1-a$

Intuitively, we try to ensure that $(\boldsymbol{x}_i)_j = 1$ if $\pi_i \models \phi_j$ and $(\boldsymbol{x}_i)_j = 0$ otherwise. We use the conversions from formal logic to soften logic shown in Table 1. The computation of $(\boldsymbol{x}_i)_j$ is based on the semantics of satisfaction relation. Specifically, $\boldsymbol{\theta}_{atom}$ is used to select atomic propositions, whereas $\boldsymbol{\theta}_{\circ}$ for $\circ \in \{\neg, \wedge, \bigcirc, \mathcal{U}\}$ is used to select operators.

According to Definition 1, whether $\pi_i \models \phi_j$ can be checked as follows. If $\phi_j = p_k$ for some $p_k \in \mathbb{P}$ (*i.e.*, $(\boldsymbol{\theta}_{atom})_{j,k} = 1$), then $\pi_i \models \phi_j$ if p_k is true at time i (*i.e.*, $(\boldsymbol{s})_{i,k} = 1$). If $\phi_j = \neg\phi_{j+1}$ (*i.e.*, $(\boldsymbol{\theta}_{\neg})_j = 1$), then $\pi_i \models \phi_j$ if and only if $\pi_i \not\models \phi_{j+1}$, *i.e.*, $(\boldsymbol{x}_i)_j = 1-(\boldsymbol{x}_i)_{j+1}$. If $\phi_j = \phi_{j+1} \wedge \phi_k$ (*i.e.*, $(\boldsymbol{\theta}_{\wedge})_j = 1$), then $\pi_i \models \phi_j$ if and only if $\pi_i \models \phi_{j+1}$ and $\pi_i \models \phi_k$, *i.e.*, $(\boldsymbol{x}_i)_j = \sigma((\boldsymbol{x}_i)_{j+1} + (\boldsymbol{r}_i)_j - 1)$ where $(\boldsymbol{r}_i)_j$ is used to compute whether the right sub-formula ϕ_k of ϕ_j fulfills $\pi_i \models \phi_k$. If $\phi_j = \bigcirc\phi_{j+1}$ (*i.e.*, $(\boldsymbol{\theta}_{\bigcirc})_j = 1$), then $\pi_i \models \phi_j$ if and only if $\pi_{i+1} \models \phi_{j+1}$, *i.e.*, $(\boldsymbol{x}_i)_j = (\boldsymbol{x}_{i+1})_{j+1}$. If $\phi_j = \phi_{j+1} \,\mathcal{U}\, \phi_k$ (*i.e.*, $(\boldsymbol{\theta}_{\mathcal{U}})_j = 1$), then whether $\pi_i \models \phi_j$ is determined by two cases. In the first case, if $\pi_i \models \phi_k$ (*i.e.*, $(\boldsymbol{r}_i)_j = 1$), then $\pi_i \models \phi_j$. In the second case, if $\pi_i \models \phi_{j+1}$ (*i.e.*, $(\boldsymbol{x}_i)_{j+1} = 1$) and $\pi_{i+1} \models \phi_j$ (*i.e.*, $(\boldsymbol{x}_{i+1})_j = 1$), then $\pi_i \models \phi_j$. Note that in the second case, since there is a loop in π and the computation of $(\boldsymbol{x}_i)_j$ depends on the result $(\boldsymbol{x}_{i+1})_j$ at the next time, the computation of $(\boldsymbol{x}_i)_j$ requires iterative computation of $(\boldsymbol{u}_i)_j^{(t)}$ for t increases from 1 to $|\pi|$. We use Example 3 to illustrate Definition 3.

Example 3. *Let π, ϕ, $(\boldsymbol{s}, \boldsymbol{l})$, and η be given by Example 1 and Example 2, then $|\phi| = 4$ and $|\pi| = 2$. The parameters of NTCNet are as follows: $(\boldsymbol{\theta}_{\mathcal{U}})_1 = 1$, $(\boldsymbol{\theta}_{atom})_{2,1} = 1$, $(\boldsymbol{\theta}_{\bigcirc})_3 = 1$, $(\boldsymbol{\theta}_{atom})_{4,2} = 1$, $(\boldsymbol{\theta}_{right})_{1,3} = 1$, and all other elements of θ are 0. We compute NTCNet$((\boldsymbol{s}, \boldsymbol{l}) \,|\, \eta)$ as follows. For $j = 4$, $(\boldsymbol{x}_1)_4 = (\boldsymbol{\theta}_{atom})_{4,2}(\boldsymbol{s})_{1,2} = 1$ and $(\boldsymbol{x}_2)_4 = (\boldsymbol{\theta}_{atom})_{4,2}(\boldsymbol{s})_{2,2} = 0$. For $j = 3$, $(\boldsymbol{x}_1)_3 = (\boldsymbol{\theta}_{\bigcirc})_3(\boldsymbol{x}_2)_4 = 0$ and $(\boldsymbol{x}_2)_3 = (\boldsymbol{\theta}_{\bigcirc})_3(\boldsymbol{x}_3)_4 = (\boldsymbol{\theta}_{\bigcirc})_3(\boldsymbol{l})_1(\boldsymbol{x}_1)_4 = 1$. For $j = 2$, $(\boldsymbol{x}_1)_2 = (\boldsymbol{\theta}_{atom})_{2,1}(\boldsymbol{s})_{1,1} = 1$ and $(\boldsymbol{x}_2)_2 = (\boldsymbol{\theta}_{atom})_{2,1}(\boldsymbol{s})_{2,1} = 1$. For $j = 1$, the computation of $(\boldsymbol{u}_i)_1^{(t)}$ for t increases from 1 to $|\pi| = 2$ is needed. Firstly $(\boldsymbol{u}_1)_1^{(1)} = (\boldsymbol{r}_1)_1 = (\boldsymbol{\theta}_{right})_{1,3}(\boldsymbol{x}_1)_3 = 0$, $(\boldsymbol{u}_2)_1^{(1)} = (\boldsymbol{r}_2)_1 = (\boldsymbol{\theta}_{right})_{1,3}(\boldsymbol{x}_2)_3 = 1$, and $(\boldsymbol{u}_3)_1^{(1)} = (\boldsymbol{l})_1(\boldsymbol{u}_1)_1^{(1)} = 0$. Secondly, $(\boldsymbol{u}_1)_1^{(2)} = \sigma(\sigma((\boldsymbol{x}_1)_2 + (\boldsymbol{u}_2)_1^{(1)} - 1) + (\boldsymbol{u}_1)_1^{(1)}) = 1$ and $(\boldsymbol{u}_2)_1^{(2)} = \sigma(\sigma((\boldsymbol{x}_2)_2 + (\boldsymbol{u}_3)_1^{(1)} - 1) + (\boldsymbol{u}_2)_1^{(1)}) = 1$. Finally, $(\boldsymbol{x}_1)_1 = (\boldsymbol{\theta}_{\mathcal{U}})_1(\boldsymbol{u}_1)_1^{(2)} = 1$ and $(\boldsymbol{x}_2)_1 = (\boldsymbol{\theta}_{\mathcal{U}})_1(\boldsymbol{u}_2)_1^{(2)} = 1$. Therefore, NTCNet$((\boldsymbol{s}, \boldsymbol{l}) \,|\, \eta) = (\boldsymbol{x}_1)_1 = 1$.*

4.3 Correctness of Neural Trace Checking

We provide a theoretical result in Theorem 1 to bridge LTL trace checking to neural trace checking. The proof of Theorem 1 is not straightforward because it needs to compute the semantics of the operator $\mathcal{U}$ on *infinite traces.* [2] We first introduce several lemmas to bridge the connection between $\boldsymbol{u}$ and the semantic of the operator $\mathcal{U}$. Lemma 1 shows a well-known equivalent semantic of the operator $\mathcal{U}$, which implies that checking the satisfaction relation between π and an LTL formula $\phi_j \,\mathcal{U}\, \phi_k$ on a time point can be reduced to checking on subsequent time points.

Lemma 1. *Let ϕ_i be an LTL formula of the form $\phi_j \,\mathcal{U}\, \phi_k$. Then, $\phi_i \equiv \phi_k \vee (\phi_j \wedge \bigcirc\phi_i)$.*

The next two lemmas demonstrate two important properties of the calculation of $\boldsymbol{u}$. In Lemma 2, we show that $(\boldsymbol{u}_i)_j^{(t)} = 1$, if

[2]The proofs of Theorem 1 and its supporting lemmas can be found in the technical report: https://github.com/sysulic/VSCNet.

there exists a time i' where $0 \leq i' - i < t$ such that $\pi_{i'} \models \phi_k$ and $\pi_{i''} \models \phi_{j+1}$ for all $i'' \in [i, i')$. Additionally, Lemma 3 shows that $\boldsymbol{u}$ is monotonically increasing with time points, *i.e.*, $(\boldsymbol{u}_i)_j^{(t)} = 1$ implies $(\boldsymbol{u}_i)_j^{(t+1)} = 1$. This implies that computing the vector $\boldsymbol{u}_i^{(t)}$ for t increases from 1 to $|\pi|$ is sufficient to determine the satisfaction relation between a finite trace π and $\phi_j \,\mathcal{U}\, \phi_k$.

Lemma 2. *Let π be a trace and $\phi_j = \phi_{j+1} \,\mathcal{U}\, \phi_k$, where ϕ_j, ϕ_{j+1}, and ϕ_k are LTL formulae. Supposed that $(\boldsymbol{u}_i)_j^{(1)} \Leftrightarrow (\boldsymbol{x}_i)_k$, $(\boldsymbol{x}_i)_k = 1 \Leftrightarrow \pi_i \models \phi_k$, and $(\boldsymbol{x}_i)_{j+1} = 1 \Leftrightarrow \pi_i \models \phi_{j+1}$ for every $i \in [1, |\pi|]$. Let $i' = i + t$ where $t \geq 0$. For every $i \in [1, |\pi| + 1]$, if $(\pi_{i'} \models \phi_k) \wedge \forall i'' \in [i, i')(\pi_{i''} \models \phi_{j+1})$ holds, then $(\boldsymbol{u}_i)_j^{(t+1)} = 1$.*

Lemma 3. *For any $i \in [1, |\pi|]$, $(\boldsymbol{u}_i)_j^{(t)} = 1$ only if $(\boldsymbol{u}_i)_j^{(t+1)} = 1$, where $t \geq 1$.*

Theorem 1 shows that NTCNet is able to check the satisfaction relation between a possibly infinite trace π and an LTL formula ϕ. Based on the syntax tree of ϕ, the proof follows the computation of the satisfaction vector $(\boldsymbol{x}_1)_1$. The most important step lies in inferring the satisfaction relation between π and an LTL formula ϕ_j of the form $\phi_{j+1} \,\mathcal{U}\, \phi_k$. In this step the computation of $(\boldsymbol{u}_1)_j^{(|\pi|)}$ suffices to check whether $\pi \models \phi_j$.

Theorem 1. *Let ϕ be an LTL formula and η its LTL encoding. For any trace π,* NTCNet(*TENSORIZE*(π)|η) = 1 *if and only if* $\pi \models \phi$.

5 Neural Modeling for SAT-and-GET

Theorem 1 enables a new way to tackle the problem of checking LTL satisfiability and generating traces as evidence, namely the SAT-and-GET problem, since the implicit input of NTCNet can be used as weak supervision to guide a neural network to generate satisfiable traces from a satisfiable LTL formula. In this section, we first overview the neural model VSCNet for tackling the SAT-and-GET problem, then introduce its two main modules – SCNet for checking LTL satisfiability and TGNet for generating traces.

5.1 Overview of VSCNet

The module SCNet aims to extract the feature vector $\boldsymbol{\phi}$ for representing a given LTL formula ϕ and to predict the probability p_{SC} that ϕ is satisfiable based on $\boldsymbol{\phi}$. By SCNet(ϕ) we denote the output of SCNet for the input formula ϕ, which is $(p_{SC}, \boldsymbol{\phi})$. The module TGNet exploits $\boldsymbol{\phi}$ to generate a tensorized trace $(\boldsymbol{s}, \boldsymbol{l})$. By TGNet($\boldsymbol{\phi}$) we denote the output of TGNet for the input feature vector $\boldsymbol{\phi}$, which is $(\boldsymbol{s}, \boldsymbol{l})$. With NTCNet, we are able to design a loss function in Equation (6) for jointly training of SCNet and TGNet.

$$\mathcal{L} = \mathcal{L}_{SC} + \lambda \mathcal{L}_{TG} \tag{6}$$

where $\lambda \in \mathbb{R}$ is a hyperparameter used to determine the relative importance of VSCNet on trace generation, and $\mathcal{L}_{SC}$ and $\mathcal{L}_{TG}$ are the losses for SCNet and TGNet, respectively, defined in Section 5.2 and Section 5.3.

Figure 1 gives an overview of training VSCNet. Based on two SOTA representation learning methods for LTL formulae [36, 37], we accordingly design two different implementations of VSCNet, namely VSCNet-T and VSCNet-G.

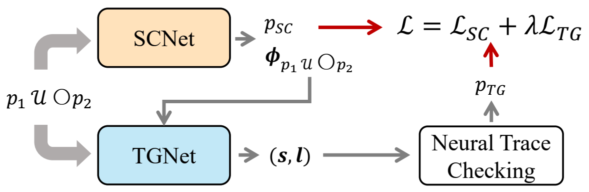

SCNet
p_{SC}
$\boldsymbol{\phi}_{p_1 \mathcal{U} \bigcirc p_2}$
$\mathcal{L} = \mathcal{L}_{SC} + \lambda \mathcal{L}_{TG}$
$p_1 \,\mathcal{U} \bigcirc p_2$
p_{TG}
TGNet
$(\boldsymbol{s}, \boldsymbol{l})$
Neural Trace Checking

Figure 1: The overview of training VSCNet.

```
Algorithm 3: Solving the SAT-and-GET problem
  Input: An LTL formula $\phi$, and two threshold
         hyperparameters $\beta_{\top}$ for finding tensorized traces
         and $\beta_{SAT}$ for determining the satisfiability of $\phi$.
  Output: Whether $\phi$ is satisfiable or not and a satisfiable
          trace $\pi$ of $\phi$ if it is satisfiable.
1 $(p_{SC}, \boldsymbol{\phi}) \leftarrow$ SCNet($\phi$)
2 $(\boldsymbol{s}, \boldsymbol{l}) \leftarrow$ TGNet($\boldsymbol{\phi}$)
3 $\pi \leftarrow$ ROUND($(\boldsymbol{s}, \boldsymbol{l})$, $\beta_{\top}$)
4 if $\pi \models \phi$ or $p_{SC} > \beta_{SAT}$ then
5     return SAT, $\pi$
6 else
7     return UNSAT
```

Algorithm 3 shows how to apply VSCNet to solve the SAT-and-GET problem. Specifically, VSCNet first invokes SCNet and TGNet to obtain the satisfiable probability p_{SC} of ϕ and a *soften* tensorized trace $(\boldsymbol{s}, \boldsymbol{l})$ of ϕ where all elements of $\boldsymbol{s}$ and $\boldsymbol{l}$ are real values between 0 and 1 (lines 1-2 of Algorithm 3). Then, VSCNet rounds probabilistic $(\boldsymbol{s}, \boldsymbol{l})$ to π as shown in Algorithm 4, where $\beta_{\top} \in (0, 1)$ is a hyperparameter to distinguish a true case from a false case and k in line 5 is the loop-start time. After that, VSCNet checks the satisfaction relation between π and ϕ either by NTCNet or by a classical trace checking algorithm. In our experiments, we employ the classical algorithm proposed in [39] for such a checking since the classical algorithm is more efficient when solving a single formula. Finally, VSCNet returns the satisfiability conclusion of ϕ and if ϕ is satisfiable, the generated trace π as well (lines 4-7 of Algorithm 3).

```
Algorithm 4: ROUND
  Input: A soften tensorized trace $(\boldsymbol{s}, \boldsymbol{l})$, where $\boldsymbol{s} \in \mathbb{R}^{m\times n}_{[0,1]}$ and
         $\boldsymbol{l} \in \mathbb{R}^{m}_{[0,1]}$, and a hyperparameter $\beta_{\top}$ for rounding
         elements in $\boldsymbol{s}$.
  Output: A trace $\pi$.
1 $\pi[i] \leftarrow \emptyset$ for each $(\boldsymbol{s})_i$
2 for each $(\boldsymbol{s})_{i,j} \in \boldsymbol{s}$ do
3     if $(\boldsymbol{s})_{i,j} > \beta_{\top}$ then
4         $\pi[i] \leftarrow \pi[i] \cup \{p_j\}$
5 $k \leftarrow \arg\max_i((\boldsymbol{l})_i)$
6 $\pi \leftarrow \pi[1], \ldots, (\pi[k], \ldots, \pi[m])^{\omega}$
7 return $\pi$
```

5.2 SCNet: Checking LTL Satisfiability

To check the satisfiability of an LTL formula, we implement SCNet-T and SCNet-G based on two SOTA representation learning methods for LTL formulae [36, 37].

5.2.1 SCNet-T. We first utilize the extractor TreeNN-inv [36] to obtain an embedding of an LTL formula. Then, a fully-connected layer is applied to the embedding to obtain the satisfiable probability of the formula. Formally, SCNet-T is defined as follows:

$$p_{SC} = (\texttt{softmax}(\texttt{ReLU}(\boldsymbol{W}_1\boldsymbol{\phi} + \boldsymbol{b}_1)))_2, \tag{7}$$

where $\boldsymbol{\phi} \in \mathbb{R}^{d_m}$ is the embedding generated by Algorithm 1 for the given LTL formula, $\boldsymbol{W}_1 \in \mathbb{R}^{2\times d_m}$, $\boldsymbol{b}_1 \in \mathbb{R}^2$ are the learnable weight and bias, softmax is the softmax function, and where d_m is a hyperparameter used in Algorithm 1. We measure how well SCNet-T checks the satisfiability using the cross-entropy loss $\mathcal{L}_{SC}$ between p_{SC} and the ground truth.

5.2.2 SCNet-G. Different from SCNet-T, after constructing the one-step unfolded graph (OSUG) [37] of the given LTL formula, we compute the graph embedding $\boldsymbol{v}_g^{(T)} \in \mathbb{R}^{d_h}$ and vertex embeddings $\boldsymbol{v}_i^{(T)} \in \mathbb{R}^{d_h}$ for all vertices v_i in OSUG, using a 10-layer OSUG-based extractor [37] with hidden layers of dimension d_h and initial embeddings of dimension d_m. The satisfiable probability and the loss are defined in the same way as that of SCNet-T, where $\boldsymbol{\phi}$ in Equation (7) is set to $\boldsymbol{v}_g^{(T)}$.

5.3 TGNet: Generating a Satisfiable Trace

Theorem 1 inspires us to design a neural network – TGNet, which is enforced to generate a trace as satisfiable as possible through NTCNet. Due to differences among LTL representations [36, 37], we design TGNet-T and TGNet-G for SCNet-T and SCNet-G, respectively.

5.3.1 TGNet-T. We iteratively generate a trace state by state. At each iteration i ($1 \le i \le N_t$, where N_t is a hyperparameter representing the trace size), we utilizes a two-layer MLP to predict the probability $(\boldsymbol{s})_{i,j} \in \mathbb{R}_{[0,1]}$ that atomic proposition p_j is true at time i, defined as follows:

$$(\boldsymbol{s})_i = \texttt{sigmoid}(\boldsymbol{W}_3\texttt{ReLU}(\boldsymbol{W}_2[\boldsymbol{\phi}, (\boldsymbol{s})_{i-1}] + \boldsymbol{b}_2) + \boldsymbol{b}_3), \tag{8}$$

where $\boldsymbol{W}_2 \in \mathbb{R}^{d_h\times(d_m+N_a)}$, $\boldsymbol{b}_2 \in \mathbb{R}^{d_h}$, $\boldsymbol{W}_3 \in \mathbb{R}^{N_a\times d_h}$, $\boldsymbol{b}_3 \in \mathbb{R}^{N_a}$ are learnable weights and biases, and sigmoid is the sigmoid function. Besides of N_t, N_a and d_h are also hyperparameters, where N_a denotes the number of atomic propositions. $[\boldsymbol{\phi}, (\boldsymbol{s})_{i-1}]$ denotes the concatenation of $\boldsymbol{\phi}$ and $(\boldsymbol{s})_{i-1}$. Specially, $(\boldsymbol{s})_0 = \boldsymbol{0}$. After N_t iterations, we obtain $\boldsymbol{s}$, and then predict the loop-start time, defined as follows:

$$\begin{aligned} \boldsymbol{l} &= \texttt{softmax}([x_1, \dots, x_{N_t}]), \\ x_i &= \boldsymbol{W}_5\texttt{ReLU}(\boldsymbol{W}_4[\boldsymbol{\phi}, (\boldsymbol{s})_i, (\boldsymbol{s})_{N_t}] + \boldsymbol{b}_4) + \boldsymbol{b}_5, \end{aligned} \tag{9}$$

where $\boldsymbol{W}_4 \in \mathbb{R}^{d_h\times(d_m+N_a+N_a)}$, $\boldsymbol{b}_4 \in \mathbb{R}^{d_h}$, $\boldsymbol{W}_5 \in \mathbb{R}^{1\times d_h}$, $\boldsymbol{b}_5 \in \mathbb{R}$ are learnable weights and biases. We measure how well TGNet-T does by the following loss function:

$$\mathcal{L}_{TG} = \frac{1}{|D|}\sum_{(x,y)\in D}(\texttt{NTCNet}((\boldsymbol{s}_x, \boldsymbol{l}_x)\,|\,\eta_x) - y)^2, \tag{10}$$

where D is the training set, x is an LTL formula, y is the ground truth of x, $(\boldsymbol{s}_x, \boldsymbol{l}_x)$ is the softened tensorized trace of x generated by TGNet-T, and η_x is the LTL encoding of x. The ground truth of x is defined as: if x is satisfiable, then $y = 1$, otherwise $y = 0$.

In the course of training, to overcome the problem of vanishing gradient, we use a variant of leaky ReLU defined in Equation (11) as a differentiable approximation of the activation function $\sigma(x)$.

$$\widehat{\sigma}(x) = \begin{cases} \alpha x, & x < 0, \\ x, & 0 \le x \le 1, \\ \alpha x + 1 - \alpha, & x > 1, \end{cases} \tag{11}$$

where $\alpha \in \mathbb{R}$ is a hyperparameter.

5.3.2 TGNet-G. We also iteratively generate a trace state by state. At each iteration i ($1 \le i \le N_t$), we first extract the feature vector $\boldsymbol{v}_j^{(T,i)} \in \mathbb{R}^{d_h}$ for each vertex v_j in OSUG via the one-layer OSUG-based extractor [37] with hidden layers of dimension d_h and initial embeddings of dimension d_h defined as follows:

$$\begin{aligned} \boldsymbol{v}_j^{(T,i)} &= \texttt{ReLU}(\boldsymbol{W}_6\boldsymbol{v}_j^{(T,i-1)} + \boldsymbol{W}_7\tfrac{1}{|\mathsf{N}(j)|}\textstyle\sum_{k\in\mathsf{N}(j)}\boldsymbol{v}_k^{(T,i-1)}), \\ \boldsymbol{v}_j^{(T,0)} &= \boldsymbol{v}_j^{(T)}, \end{aligned} \tag{12}$$

where $\boldsymbol{v}_j^{(T,0)}$ is the initial embedding of v_j and $\boldsymbol{W}_6, \boldsymbol{W}_7 \in \mathbb{R}^{d_h\times d_h}$ are learnable weights. We set $\boldsymbol{v}_j^{(T,0)}$ to the vertex embedding of v_j computed by SCNet-G. Then, the vertex embeddings corresponding to atomic propositions are fed to a classifier to obtain the probability that the atomic propositions are true at time i. The classifier is a two-layer MLP defined as follows:

$$(\boldsymbol{s})_{i,j} = (\texttt{softmax}(\texttt{ReLU}(\boldsymbol{W}_9\texttt{ReLU}(\boldsymbol{W}_8\boldsymbol{v}_j^{(T,i)} + \boldsymbol{b}_8) + \boldsymbol{b}_9)))_2, \tag{13}$$

where $\boldsymbol{W}_8 \in \mathbb{R}^{d_h\times d_h}$, $\boldsymbol{b}_8 \in \mathbb{R}^{d_h}$, $\boldsymbol{W}_9 \in \mathbb{R}^{2\times d_h}$, $\boldsymbol{b}_9 \in \mathbb{R}^2$ are learnable weights and biases. We compute graph embeddings $\boldsymbol{v}_g^{(T,i)}$ for $1 \le i \le N_t$ and predict the loop-start time defined as follows:

$$\begin{aligned} \boldsymbol{l} &= \texttt{softmax}([x_1, \dots, x_{N_t}]), \\ x_i &= \boldsymbol{W}_{11}\texttt{ReLU}(\boldsymbol{W}_{10}\boldsymbol{v}_g^{(T,i)} + \boldsymbol{b}_{10}) + \boldsymbol{b}_{11}, \end{aligned} \tag{14}$$

where $\boldsymbol{W}_{10} \in \mathbb{R}^{d_h\times d_h}$, $\boldsymbol{b}_{10} \in \mathbb{R}^{d_h}$, $\boldsymbol{W}_{11} \in \mathbb{R}^{1\times d_h}$, $\boldsymbol{b}_{11} \in \mathbb{R}$ are learnable weights and biases. The computation of $\mathcal{L}_{TG}$ and the differentiable approximation of the activation function are the same as that of TGNet-T.

5.4 Highlights of VSCNet

Given an LTL formula ϕ, VSCNet returns a satisfiability conclusion and possibly an additional trace π. In this way VSCNet ensures the soundness of the satisfiability conclusion if $\pi \models \phi$. In the case where $\pi \models \phi$, if SCNet answers that ϕ is satisfiable, then π confirms such an answer, otherwise π is able to correct the answer of SCNet. It strengthens the credibility of a neural network-based approach. In contrast, most of the existing neural network-based approaches [36] only consider binary classification and lack such a credibility. On the other hand, when $\pi \not\models \phi$, the answer of VSCNet is just a prediction of neural networks, which is not verifiable. Improving the soundness in this case will be our future work.

The inference time of both SCNet and TGNet is polynomial in terms of the size of the input formula, thus we are able to check the satisfiability of an LTL formula in polynomial time, which cannot be guaranteed by logic-based approaches. Although the inference time does not include the training time, model training can be done offline and once. In the case where the trained model has high performance and good generalizability, our proposed approach can directly be used to check the satisfiablity of an arbitrary LTL formula without retraining.

6 Experiment

We conducted experiments to answer four research questions:

RQ 1. *How well does* `VSCNet` *work on different datasets?*

RQ 2. *How well does* `VSCNet` *generalize across formula sizes?*

RQ 3. *How well is* `VSCNet` *compared with SOTA logic-based approaches?*

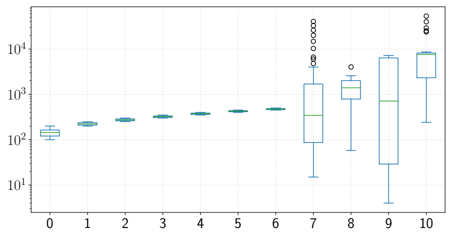
RQ 4. *Is every module of* `VSCNet` *designed reasonably?*

6.1 Competitor

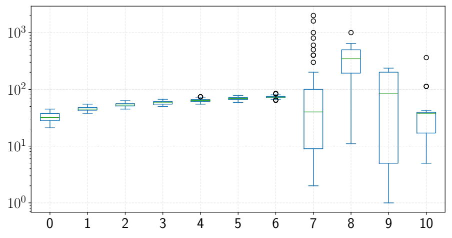
The competitors include seven neural network-based approaches.
`TreeNN-inv`, `TreeNN-MP`, `TreeNN-con` [36]. They achieve the SOTA performance on large-scale datasets in checking LTL satisfiability since their architecture matches the logical properties of LTL [36]. However, they cannot generate traces.
`Transformer-SC`, `Transformer-TG`, `Transformer`. `Transformer-SC` [36] employs the encoder of Transformer [58] to compute the embedding of an LTL formula, and feeds the embedding into an MLP to obtain the satisfiable probability. `Transformer-TG` [21] employs the encoder and decoder of Transformer to generate a trace token-by-token for a satisfiable formula. But it cannot check whether a formula is satisfiable. `Transformer` is a simple combination of `Transformer-SC` and `Transformer-TG`. It is enhanced from `Transformer-TG` by also feeding the embedding obtained by the encoder into an MLP to obtain the satisfiable probability.
`OSUG` [37]. It is a SOTA approach to solving the SAT-and-GET problem. It exploits OSUG to represent LTL formulae and trains two neural networks to generate satisfiable traces based on RL and to check satisfiability based on MLP, respectively.

To demonstrate the advantage of our approach in terms of efficiency, we also compared our approach with two logic-based approaches: `nuXmv` and `Aalta`, following the setups in [36, 37]. As empirically demonstrated by [30], neither of these two approaches always outperforms the opposite approach.
`nuXmv` [10]. It is one of the SOTA approaches to model checking. Since LTL satisfiability checking can be reduced to model checking [45], `nuXmv` has also been compared with existing approaches to LTL satisfiability checking [30, 36].
`Aalta` [29, 30]. It employs a SAT solver to conduct LTL satisfiability checking. Thanks to the development of modern SAT solvers, it becomes one of the SOTA approaches to checking LTL satisfiability.

Logic-based approaches are sound and complete, but they all suffer from the search space explosion problem. Compared with logic-based approaches, our approach is approximate but is guaranteed to check LTL satisfiability in polynomial time. Since the theoretical worst case is rare in practice, we need to compare our approach with logic-based approaches to confirm the efficiency advantage. Besides, we reported the performance of a random approach (denoted 'random') which makes random guesses in checking satisfiability and in generating traces. It helps to highlight the difficulty in solving the SAT-and-GET problem and the effectiveness of a neural network-based approach.



(a) Formula size.



(b) Number of atomic propositions.

Figure 2: The boxplots of the formula size and the number of atomic propositions, where '0' to '10' correspond to *SPOT*-[100, 200), -[200, 250), -[250, 300), -[300, 350), -[350, 400), -[400, 450), -[450, 500), *LTL-as-LTL$_f$*, *LTL$_f$-Specific*, *NASA-Boeing*, and *DECLARE* respectively.

6.2 Dataset

We used synthetic datasets *SPOT*-$[n, m)$ proposed by [36] to train and test neural networks, where $[n, m)$ is the interval of the formula size. Besides, we followed [30, 36] to evaluate our approaches on the four large-scale datasets: *LTL-as-LTL$_f$* [50], *LTL$_f$-Specific* [11, 28], *NASA-Boeing* [8, 20], and *DECLARE* [18]. Among them, a portion of formulae in *LTL-as-LTL$_f$* [2, 7, 32, 63], *NASA-Boeing*, and *DECLARE* are derived from industry. Satisfiability checking of these formulae supports several real applications, such as a step that should take place before verifying reactive systems to assure the consistency of temporal specifications [30].

Figure 2 shows that the formula size and the number of atomic propositions in formulae in large-scale datasets are usually larger than that in synthetic datasets. In addition, the formulae in large-scale datasets come from practical applications or are randomly generated with certain templates, while the formulae in synthetic datasets are randomly generated based on a certain distribution. Therefore, large-scale datasets can be considered to have distributions different from that of synthetic datasets [36].

6.3 Setup

We followed the setups in [36]. Specifically, we trained all neural networks on the training set of *SPOT*-[100, 200) with hyperparameters optimized on the validation set of *SPOT*-[100, 200). The training

Table 2: Evaluation results on the test set of *SPOT*-[100, 200), where 'acc.', 'pre.', 'rec.', 'F1', and 'ass.' are shown in percentages (%) and 'time' denotes the sum of inference time for all formulae in seconds (in timeout cases the inference time is estimated as 3600s). Bold numbers mark the best results in neural network-based approaches. If a metrics cannot be applied to an approach, the result is marked with '-'.

approach	acc.	pre.	rec.	F1	sacc.	time
random	50.01	50.01	49.28	49.64	51.88	0
`Aalta`	100.00	100.00	100.00	100.00	100.00	12,784,302
`nuXmv`	100.00	100.00	100.00	100.00	100.00	6,848
`TreeNN-inv`	93.27	97.75	88.58	92.94	-	434
`TreeNN-MP`	86.54	90.74	81.40	85.82	-	455
`TreeNN-con`	89.38	95.96	82.22	88.56	-	459
`Transformer-SC`	72.56	74.24	69.10	71.58	-	**104**
`Transformer-TG`	-	-	-	-	47.67	5,484
`Transformer`	59.30	62.79	45.64	52.86	45.09	10,601
`OSUG`	98.47	99.15	97.79	98.46	55.22	2,816
`VSCNet-T` (our)	93.45	97.47	89.22	93.16	71.86	364
`VSCNet-G` (our)	**99.15**	**99.47**	**98.83**	**99.15**	**91.01**	**156**

of `Transformer` and `Transformer-TG` requires not only LTL formulae and their satisfiability conclusions, but also a satisfiable trace for each formula. We used `nuXmv` to compute satisfiable traces for all training sets. The training of other neural networks requires only LTL formulae and their satisfiability conclusions. We evaluated all neural networks on the test set of *SPOT*-[100, 200) to demonstrate their performance on in-distribution datasets. We also tested the generalizability of neural networks on out-of-distribution datasets, by evaluating all neural networks on the *SPOT* test sets with larger formulae and on the large-scale datasets. This generalizability test is useful for cases where the distribution is not clear or the amount of data is small. The logic-based approaches are also evaluated on the above datasets. Since `TreeNN-inv`, `TreeNN-MP`, `TreeNN-con` and `Transformer-SC` are not able to generate traces, we only evaluated them in LTL satisfiability checking. For the whole SAT-and-GET problem, we compared `VSCNet` with `Transformer` and `OSUG`, which are the only competitors that solve the whole problem.

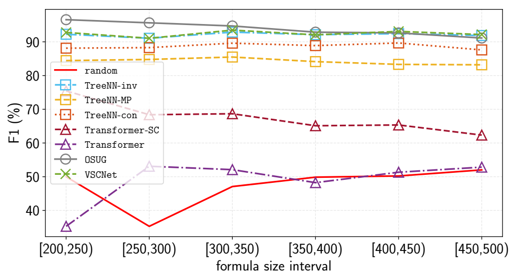
The metrics for LTL satisfiability checking include accuracy (acc.), precision (pre.), recall (rec.), and F1 score, defined as follows:

$$\begin{aligned} \text{acc.} &= \frac{\text{TP+TN}}{\text{TP+TN+FP+FN}}, & \text{pre.} &= \frac{\text{TP}}{\text{TP+FP}}, \\ \text{rec.} &= \frac{\text{TP}}{\text{TP+FN}}, & \text{F1} &= 2 \times \frac{\text{pre.}\times\text{rec.}}{\text{pre.+rec.}}, \end{aligned} \tag{15}$$

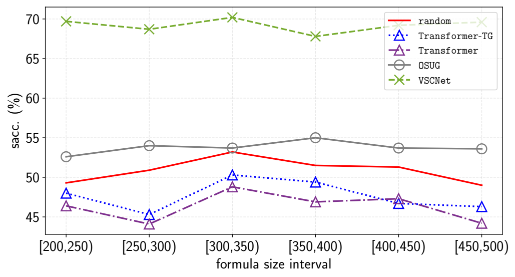
where TP, TN, FP and FN represent the numbers of true positive, true negative, false positive and false negative, respectively. Following [21], we used semantic accuracy (sacc.) to evaluate the performance of trace generation, defined as follows:

$$\text{sacc.} = \frac{|\{\phi \in D | \pi \models \phi\}|}{|D|}, \tag{16}$$

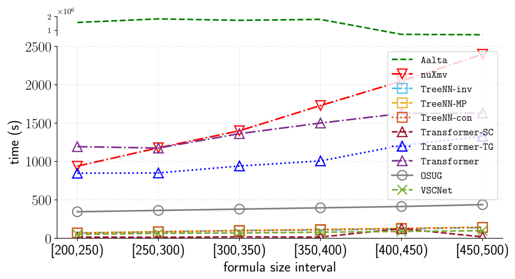
where D is the set of test formulae and π is the generated trace for a test formula ϕ. If the satisfaction relation between π and ϕ holds, the corresponding test case is semantically accurate. We employed the trace checking algorithm proposed in [39] to check whether a test case is semantically accurate. In addition, we compared the



(a) F1 score for LTL satisfiability checking.



(b) Semantic accuracy for trace generation.



(c) Inference time.

Figure 3: Evaluation results on other *SPOT* test sets. The accuracy, precision, and recall scores for all compared approaches have similar rankings and trends with the F1 score.

inference time of each approach. We set a time limit of 3600 seconds for handling each formula.

All our experiments were conducted on a Linux equipped with an Intel(R) Xeon(R) Gold 5218R processor with 2.1 GHz and 126 GB RAM. The Linux version is Ubuntu 18.04.6 LTS (GNU/Linux 5.4.0-144-generic x86_64). We trained and tested all neural networks on a single NVIDIA A100 GPU with 40GB RAM. To guarantee fairness in summing the inference time for all test formulae, we enforced that only one formula was tested at a time. We trained all neural networks with the Adam optimizer. For evaluating `TreeNN-inv`, `TreeNN-MP`, `TreeNN-con` and `Transformer`, we used the optimal hyperparameters reported by Luo et al. [36]. For evaluating `Aalta` and `nuXmv`, we followed the configuration reported by Li et al. [30]. We used grid search to find the optimal hyperparameters of `VSCNet-T` and `VSCNet-G`. The optimal hyperparameters of `VSCNet-T` are set as follows: the learning rate is $1e$-3; the batch size is 1024; $N_t = 5$;

Table 3: Evaluation results on the large-scale datasets.

approach	*LTL-as-LTL$_f$*				*LTL$_f$-Specific*				*NASA-Boeing*				*DECLARE*			
	acc.	F1	sacc.	time	acc.	F1	sacc.	time	acc.	F1	sacc.	time	acc.	F1	sacc.	time
random	50.75	65.46	54.96	0	51.59	47.88	92.96	0	62.90	77.23	24.19	0	53.21	69.46	0.00	0
Aalta	100.00	100.00	100.00	122,682	100.00	100.00	100.00	36,027	100.00	100.00	100.00	3,601	100.00	100.00	100.00	61,220
nuXmv	100.00	100.00	100.00	6,804	100.00	100.00	100.00	766	100.00	100.00	100.00	41	100.00	100.00	100.00	3,743
TreeNN-inv	88.05	93.53	-	453	**98.53**	**98.37**	-	305	83.87	91.23	-	33	78.38	87.88	-	176
TreeNN-MP	82.18	90.03	-	480	96.29	95.98	-	234	70.97	83.02	-	25	95.50	97.70	-	129
TreeNN-con	84.61	91.10	-	475	98.35	98.11	-	230	51.61	68.09	-	25	0.00	0.00	-	129
Transformer-SC	72.99	82.64	-	**161**	83.12	75.45	-	**55**	71.05	83.08	-	**3**	0.00	0.00	-	**3**
OSUG	8.62	15.87	55.26	11,444	55.71	71.55	93.09	5,173	3.23	6.25	**46.77**	341	10.09	18.33	0.00	15,730
VSCNet-T (our)	88.35	**93.71**	**81.46**	368	90.76	90.56	83.80	177	**90.32**	**94.92**	**46.77**	21	95.41	97.65	0.00	94
VSCNet-G (our)	**88.85**	94.03	51.68	**91**	45.94	61.66	**93.36**	**25**	83.87	91.23	43.55	**3**	**100.00**	**100.00**	0.00	**6**

$N_a = 1024$; $d_m = 256$; $d_h = 128$; $\lambda = 0.01$; $\beta_{SAT} = 0.6$; $\beta_{\top} = 0.6$. The optimal hyperparameters of VSCNet-G are set as follows: the learning rate is 1e-3; the batch size is 1024; $N_t = 5$; $d_m = 256$; $d_h = 512$; $\lambda = 0.01$; $\beta_{SAT} = 0.6$; $\beta_{\top} = 0.6$.

6.4 Comparison on Synthetic Datasets

6.4.1 LTL Satisfiability Checking. Table 2 shows that VSCNet-G achieves the best performance on the test set of *SPOT*-[100, 200). VSCNet-T has the same feature extractor as TreeNN-inv, but it slightly outperforms TreeNN-inv, indicating that the additional trace generation is helpful for satisfiability checking. On the other hand, although OSUG is also able to check LTL satisfiability and to generate traces, it uses different neural networks without interaction for these two tasks. The performance superiority of VSCNet-G beyond OSUG confirms that the interaction of the module for satisfiability checking and the module for trace generation further improves the performance for satisfiability checking. VSCNet-G outperforms VSCNet-T mainly because of the OSUG-based extractor. This reason is consistent with the findings drawn by [37].

Figure 3 (a) shows that VSCNet-G and VSCNet-T outperform all other approaches, achieving rather stable F1 scores against different formula sizes. This indicates that VSCNet-G and VSCNet-T have a good generalizability to larger formulae. The F1 score of Transformer is the lowest among all neural network-based approaches, especially lower than Transformer-SC which focuses on LTL satisfiability checking alone. It suggests that a simple combination of the module for satisfiability checking and the module for trace generation may not take effect.

6.4.2 Trace Generation. Table 2 shows that both VSCNet-T and VSCNet-G outperform all other competitors, at least achieving absolute 16.64% and 35.79% improvements in semantic accuracy, respectively. Besides, for out-of-distribution test sets, VSCNet-T and VSCNet-G still achieve significant improvements compared with all other competitors, as shown in Figure 3 (b). Figure 3 (b) also demonstrates a good generalizability of VSCNet-T and VSCNet-G against different formula sizes.

Note that Transformer-TG and Transformer perform even worse than random guessing, which is inconsistent with the results reported in [21]. The reason for this contradiction is two-fold. On one hand, the search space for traces is exponential in the number of atomic propositions. More atomic propositions will make trace generation harder. This work considers averagely 34.27 atomic propositions for one formula, while Hahn et al. [21] considers 5 atomic propositions for one formula. On the other hand, in practice, the larger a formula is, the harder its satisfied traces are often generated [30]. This work considers much larger formulae than [21].

Furthermore, compared with OSUG, our approach is much easier to converge. OSUG spends 261.43 hours to converge, while both VSCNet-G and VSCNet-T spend about 6.92 hours to converge.

6.4.3 Inference Time. As shown in Table 2, the neural network-based approaches are significantly more efficient than logic-based approaches. In particular, VSCNet-G is about 35079X faster than Aalta and about 44X faster than nuXmv. Figure 3 (c) further confirms this advantage in efficiency. Although Transformer-SC has the lowest inference time, it only performs inference for a single task (satisfiability checking). If the trace generation task is considered, the inference time will become much longer. The inference time of VSCNet is much shorter than that of Transformer-TG and Transformer, since VSCNet generates a state at a time, while both Transformer-TG and Transformer generate a variable assignment at a time, and one state consists of multiple variable assignments.

6.4.4 Summary. Experimental results on different datasets demonstrate the advantages of VSCNet in both the satisfiability checking task and the trace generation task. In particular, VSCNet significantly outperforms existing neural network-based approaches in trace generation. These results answer **RQ 1**. The results also show that VSCNet is significantly more efficient than logic-based SOTA approaches while achieving comparable performance, answering **RQ 3**. Moreover, the results also demonstrate a good generalizability of VSCNet against different formula sizes, answering **RQ 2**.

6.5 Comparison on Large-Scale Datasets

Since Transformer-TG and Transformer are inferior to random guessing on most large-scale datasets, we omit their results. Table 3 shows that all neural network-based approaches exhibit varying degrees of performance fluctuations on the large-scale datasets. However, VSCNet-G and VSCNet-T are still comparable with the logic-based approaches in LTL satisfiability checking. Besides, VSCNet-G and VSCNet-T are still significantly more efficient than the logic-based approaches. For example, compared with nuXmv and Aalta, VSCNet-G achieves averagely 186X and 3541X speedups. These results further answer **RQ 2** and **RQ 3**.

Table 4: Ablation results of VSCNet-T and its variants on all *SPOT* test sets. 'w/o SS' means VSCNet-T without satisfiability supervision. This variant uses only the generated trace to check the satisfiability: it returns SAT if and only if the satisfactory relation holds. 'w STS' means VSCNet-T with satisfiable trace supervision. This variant uses a satisfiable ground-truth trace computed by nuXmv to confine the soften generated trace (s, l), minimizing the cross-entropy loss between (s, l) and the ground truth. 'w/o NTC & w STS' means a variant of VSCNet-T which exploits the satisfiable trace supervision instead of the weak supervision provided by NTCNet.

variants	[100, 200)			[200, 250)			[250, 300)			[300, 350)			[350, 400)			[400, 450)			[450, 500)		
	acc.	F1	sacc.	acc.	F1	sacc.	acc.	F1	sacc.	acc.	F1	sacc.	acc.	F1	sacc.	acc.	F1	sacc.	acc.	F1	sacc.
VSCNet-T	93.45	93.16	**71.86**	93.10	92.78	**69.70**	91.60	91.05	**68.70**	93.80	93.47	**70.20**	92.55	92.08	**67.80**	**93.45**	**93.09**	**69.20**	**92.60**	**92.16**	**69.60**
- w/o SS	77.75	71.39	55.51	75.95	68.33	51.90	77.00	70.13	54.00	76.50	69.28	53.00	77.20	70.47	54.40	75.55	67.64	51.10	75.80	68.07	51.60
- w STS	93.66	**93.37**	54.53	**93.15**	**92.81**	51.30	**92.00**	**91.50**	52.10	**94.35**	**94.07**	53.60	**93.30**	**92.90**	52.30	93.35	92.97	51.10	92.45	91.97	48.60
- w/o NTC & w STS	**93.74**	93.44	55.10	92.85	92.44	52.30	91.85	91.28	53.60	93.60	93.22	54.20	**93.30**	92.87	54.60	92.85	92.41	53.70	92.55	92.04	53.90

6.6 Ablation Study

The results for ablation study are shown in Table 4. By comparing VSCNet-T with VSCNet-T w STS, it can be seen that the supervision of satisfiable traces is harmful to generating satisfiable traces. This is expected since a satisfiable LTL formula has multiple satisfiable traces, and using one among them to confine the training will lead to some bias. Besides, the comparison results between VSCNet-T and VSCNet-T w/o NTC & w STS show that NTCNet guides trace generation better than directly using a satisfiable trace for supervision. The comparison results between VSCNet-T and VSCNet-T w/o SS demonstrate that the module for satisfiability checking in VSCNet-T is crucial to guarantee a high performance. By considering also the importance of the module for trace generation as demonstrated in Section 6.4.1, the joint learning of the two modules is shown to be mutually reinforcing. The ablation study on VSCNet-G exhibits similar results. These results answer **RQ 4**.

7 Threat to Validity

The primary threat to validity is out-of-distribution datasets. An approach achieving higher performance on datasets with varying distributions may perform better in practice because it has a better generalizability across data distributes. Our approach has a good generalizability across formula sizes, but it is still insufficient to generalize across data distributions. The adaptation of a neural network-based approach to varying data distributions is still an open problem. In future work, we plan to explore parameter-efficient fine-tuning [65] to improve the generalizability.

The trace size is also a threat to validity. Our approach only generates traces with a fixed trace size, which is given as a hyperparameter N_t. It makes our approach hard to scale to the LTL formulae that only have extremely long satisfiable traces. In practice, however, it may be sufficient to set N_t to a small constant for most cases. In our experiments, we find that all satisfiable formulae in our synthetic datasets have satisfiable traces whose sizes are no greater than 5. Besides, according to Proposition 1, our approach also works for every LTL formula that has satisfiable traces whose sizes are smaller than N_t. Nevertheless, it is still important to generate traces with varying sizes. We leave it to future work.

Proposition 1. *Let ϕ be an LTL formula and $n \in \mathbb{N}$. If there is a trace π with size $m < n$ such that $\pi \models \phi$, there must be a trace π' with size n such that $\pi' \models \phi$.*

It is straightforward to prove Proposition 1 by expanding the loop of the trace. Example 4 illustrates how to expand a trace.

Example 4. *Let $\phi = p_1 \ \mathcal{U} \ \bigcirc p_2$ be an LTL formula and $n = 3$. A satisfiable trace of ϕ is $\pi = (\{p_1, p_2\}, \{p_1\})^\omega$, where $|\pi| = 2$. We expand the loop of π to obtain $\pi' = \{p_1, p_2\}, (\{p_1\}, \{p_1, p_2\})^\omega$. Obviously, $\pi' \models \phi$ and $|\pi'| = 3$.*

8 Conclusion and Future Work

In this paper, we have proposed an effective and efficient approach to solving the whole SAT-and-GET problem. We studied a bridge between LTL trace checking and neural trace checking. This bridge enables a joint approach of LTL satisfiability checking and trace generation, which can be realized by a neural network trainable with classical gradient descent. We theoretically proved the correctness of this bridge and empirically confirmed the effectiveness of the joint approach. Overall, we believe that our work presents an important step towards LTL satisfiability checking and trace generation in polynomial time and opens up the possibility of further utilizing neural network technology in software engineering.

Future work will improve the generalizability of our approach for trace generation and demonstrate the advantages of our approach in more real applications. Besides, we will study LTL satisfiability checking with verifiable unsatisfiable results. We plan to introduce unsatisfiable cores [48, 49] as evidence for unsatisfiable results. Thus, future work will also study how to explore neural network technology to generate unsatisfiable cores.

Acknowledgments

Our experiments were conducted on an RTAI cluster, which is supported by School of Computer Science and Engineering as well as Institute of Artificial Intelligence in Sun Yat-sen University. This paper was supported by National Natural Science Foundation of China (No. 62276284, 61976232, 61876204, 51978675), Guangdong Basic and Applied Basic Research Foundation (No. 2023A1515011470, 2022A1515011355), Guangzhou Science and Technology Project (No. 202201011699), Shenzhen Science and Technology Program (KJZD20230923114059020), Guizhou Provincial Science and Technology Projects (No. 2022-259), Humanities and Social Science Research Project of Ministry of Education (No. 18YJCZH006), as well as the Fundamental Research Funds for the Central Universities, Sun Yat-sen University (No. 23ptpy31).

References

[1] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. 2019. Learning To Solve Circuit-SAT: An Unsupervised Differentiable Approach. In *ICLR*.

[2] Abdelkader Behdenna, Clare Dixon, and Michael Fisher. 2009. Deductive verification of simple foraging robotic behaviours. *Int. J. Intell. Comput. Cybern.* 2, 4 (2009), 604–643. https://doi.org/10.1108/17563780911005818

[3] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. 2017. Neural Combinatorial Optimization with Reinforcement Learning. In *ICLR*. https://doi.org/10.48550/arXiv.1611.09940

[4] Jaroslav Bendík. 2017. Consistency checking in requirements analysis. In *ISSTA*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 408–411. https://doi.org/10.1145/3092703.3098239

[5] Matteo Bertello, Nicola Gigante, Angelo Montanari, and Mark Reynolds. 2016. Leviathan: A New LTL Satisfiability Checking Tool Based on a One-Pass Tree-Shaped Tableau. In *IJCAI*. 950–956.

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *TACAS*, Vol. 1579. 193–207. https://doi.org/10.1007/3-540-49059-0_14

[7] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. 2007. Specify, Compile, Run: Hardware from PSL. In *COCV@ETAPS*, Vol. 190. 3–16. https://doi.org/10.1016/j.entcs.2007.09.004

[8] Marco Bozzano, Alessandro Cimatti, Anthony Fernandes Pires, David Jones, Greg Kimberly, T. Petri, R. Robinson, and Stefano Tonetta. 2015. Formal Design and Safety Analysis of AIR6110 Wheel Brake System. In *CAV*, Vol. 9206. 518–535. https://doi.org/10.1007/978-3-319-21690-4_36

[9] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *VMCAI*, Vol. 6538. 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

[10] 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 *CAV*. 334–342. https://doi.org/10.1007/978-3-319-08867-9_22

[11] Claudio Di Ciccio, Fabrizio Maria Maggi, and Jan Mendling. 2016. Efficient discovery of Target-Branched Declare constraints. *Inf. Syst.* 56 (2016), 258–283. https://doi.org/10.1016/j.is.2015.06.009

[12] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Roberto Sebastiani. 2002. Improving the Encoding of LTL Model Checking into SAT. In *VMCAI*, Vol. 2294. 196–207. https://doi.org/10.1007/3-540-47813-2_14

[13] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. *Formal Methods Syst. Des.* 19, 1 (2001), 7–34. https://doi.org/10.1023/A:1011276507260

[14] Matthias Cosler, Frederik Schmitt, Christopher Hahn, and Bernd Finkbeiner. 2023. Iterative Circuit Repair Against Formal Specifications. In *ICLR*. https://doi.org/10.48550/arXiv.2303.01158

[15] Renzo Degiovanni, Facundo Molina, Germán Regis, and Nazareno Aguirre. 2018. A genetic algorithm for goal-conflict identification. In *ASE*. 520–531. https://doi.org/10.1145/3238147.3238220

[16] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. 2016. Spot 2.0 - A Framework for LTL and ω -Automata Manipulation. In *ATVA*. 122–129.

[17] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. 2011. Efficient implementation of property directed reachability. In *FMCAD*. 125–134.

[18] Valeria Fionda and Gianluigi Greco. 2018. LTL on Finite and Process Traces: Complexity Results and a Practical Reasoner. *J. Artif. Intell. Res.* 63 (2018), 557–623. https://doi.org/10.1613/jair.1.11256

[19] Michael Fisher, Clare Dixon, and Martin Peim. 2001. Clausal temporal resolution. *ACM Trans. Comput. Log.* 2, 1 (2001), 12–56. https://doi.org/10.1145/371282.371311

[20] Marco Gario, Alessandro Cimatti, Cristian Mattarei, Stefano Tonetta, and Kristin Yvonne Rozier. 2016. Model Checking at Scale: Automated Air Traffic Control Design Space Exploration. In *CAV*, Vol. 9780. 3–22. https://doi.org/10.1007/978-3-319-41540-6_1

[21] Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus Norman Rabe, and Bernd Finkbeiner. 2021. Teaching Temporal Logics to Neural Networks. In *ICLR*. https://doi.org/10.48550/arXiv.2003.04218

[22] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NeurIPS*. 1024–1034.

[23] Nikolaos Karalias and Andreas Loukas. 2020. Erdos Goes Neural: an Unsupervised Learning Framework for Combinatorial Optimization on Graphs. In *NeurIPS*.

[24] Yonit Kesten, Zohar Manna, Hugh McGuire, and Amir Pnueli. 1993. A Decision Algorithm for Full Propositional Temporal Logic. In *CAV*, Vol. 697. 97–109. https://doi.org/10.1007/3-540-56922-7_9

[25] Guillaume Lample and François Charton. 2020. Deep Learning For Symbolic Mathematics. In *ICLR*. https://doi.org/10.48550/arXiv.1912.01412

[26] Jianwen Li, Geguang Pu, Lijun Zhang, Moshe Y. Vardi, and Jifeng He. 2018. Accelerating LTL satisfiability checking by SAT solvers. *J. Log. Comput.* 28, 6 (2018), 1011–1030. https://doi.org/10.1093/logcom/exy013

[27] Jianwen Li, Yinbo Yao, Geguang Pu, Lijun Zhang, and Jifeng He. 2014. Aalta: an LTL satisfiability checker over Infinite/Finite traces. In *FSE*. 731–734. https://doi.org/10.1145/2635868.2661669

[28] Jianwen Li, Lijun Zhang, Geguang Pu, Moshe Y. Vardi, and Jifeng He. 2013. LTL Satisfiability Checking Revisited. In *TIME*. 91–98. https://doi.org/10.1109/TIME.2013.19

[29] Jianwen Li, Shufang Zhu, Geguang Pu, and Moshe Y. Vardi. 2015. SAT-Based Explicit LTL Reasoning. In *HVC*, Vol. 9434. 209–224. https://doi.org/10.1007/978-3-319-26287-1_13

[30] Jianwen Li, Shufang Zhu, Geguang Pu, Lijun Zhang, and Moshe Y. Vardi. 2019. SAT-based explicit LTL reasoning and its application to satisfiability checking. *Formal Methods in System Design* 54, 2 (2019), 164–190. https://doi.org/10.1007/s10703-018-00326-5

[31] Zhaoyu Li and Xujie Si. 2022. NSNet: A General Neural Probabilistic Framework for Satisfiability Problems. In *NeurIPS*.

[32] ARM Ltd. 1999. *AMBA Specification (Rev. 2)*. www.arm.com.

[33] Weilin Luo, Pingjia Liang, Jianfeng Du, Hai Wan, Bo Peng, and Delong Zhang. 2022. Bridging LTLf Inference to GNN Inference for Learning LTLf Formulae. In *AAAI*. 9849–9857. https://doi.org/10.1609/aaai.v36i9.21221

[34] Weilin Luo, Hai Wan, Jianfeng Du, Xiaoda Li, Yuze Fu, Rongzhen Ye, and Delong Zhang. 2022. Teaching LTLf Satisfiability Checking to Neural Networks. In *IJCAI*. 3292–3298.

[35] Weilin Luo, Hai Wan, Xiaotong Song, Binhao Yang, Hongzhen Zhong, and Yin Chen. 2021. How to Identify Boundary Conditions with Contrasty Metric?. In *ICSE*. 1473–1484. https://doi.org/10.1109/ICSE43902.2021.00132

[36] Weilin Luo, Hai Wan, Delong Zhang, Jianfeng Du, and Hengdi Su. 2022. Checking LTL Satisfiability via End-to-end Learning. In *ASE*. 21:1–21:13. https://doi.org/10.1145/3551349.3561163

[37] Weilin Luo, Yuhang Zheng, Rongzhen Ye, Hai Wan, Jianfeng Du, Pingjia Liang, and Polong Chen. 2023. SAT-Verifiable LTL Satisfiability Checking via Graph Representation Learning. In *ASE*. IEEE, 1761–1765. https://doi.org/10.1109/ASE56229.2023.00173

[38] Fabrizio Maria Maggi, Marlon Dumas, Luciano García-Bañuelos, and Marco Montali. 2013. Discovering Data-Aware Declarative Process Models from Event Logs. In *BPM*, Vol. 8094. 81–96. https://doi.org/10.1007/978-3-642-40176-3_8

[39] Nicolas Markey and Philippe Schnoebelen. 2003. Model Checking a Path. In *CONCUR*, Vol. 2761. 248–262. https://doi.org/10.1007/978-3-540-45187-7_17

[40] Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *CAV*, Vol. 2725. 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

[41] Prasita Mukherjee, Haoteng Yin, Susheel Suresh, and Tiark Rompf. 2022. OCTAL: Graph Representation Learning for LTL Model Checking. *CoRR* abs/2207.11649 (2022). https://doi.org/10.48550/arXiv.2207.11649

[42] Amir Pnueli. 1977. The Temporal Logic of Programs. In *FOCS*. 46–57.

[43] Markus Norman Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. 2021. Mathematical Reasoning via Self-supervised Skip-tree Training. In *ICLR*. https://doi.org/10.48550/arXiv.2006.04757

[44] Kristin Y. Rozier and Moshe Y. Vardi. 2007. LTL Satisfiability Checking. In *SPIN*, Vol. 4595. 149–167. https://doi.org/10.1007/978-3-540-73370-6_11

[45] Kristin Y. Rozier and Moshe Y. Vardi. 2010. LTL satisfiability checking. *International Journal on Software Tools for Technology Transfer* 12, 2 (2010), 123–137. https://doi.org/10.1007/s10009-010-0140-3

[46] Frederik Schmitt, Christopher Hahn, Markus N. Rabe, and Bernd Finkbeiner. 2021. Neural Circuit Synthesis from Specification Patterns. In *NeurIPS*. 15408–15420.

[47] Viktor Schuppan. 2009. Towards a Notion of Unsatisfiable Cores for LTL. In *FSEN*, Vol. 5961. Springer, 129–145. https://doi.org/10.1007/978-3-642-11623-0_7

[48] Viktor Schuppan. 2016. Enhancing unsatisfiable cores for LTL with information on temporal relevance. *Theor. Comput. Sci.* 655 (2016), 155–192. https://doi.org/10.1016/j.tcs.2016.01.014

[49] Viktor Schuppan. 2016. Extracting unsatisfiable cores for LTL via temporal resolution. *Acta Informatica* 53, 3 (2016), 247–299. https://doi.org/10.1007/s00236-015-0242-1

[50] Viktor Schuppan and Luthfi Darmawan. 2011. Evaluating LTL Satisfiability Solvers. In *ATVA*, Vol. 6996. 397–413. https://doi.org/10.1007/978-3-642-24372-1_28

[51] Stefan Schwendimann. 1998. A New One-Pass Tableau Calculus for PLTL. In *TABLEAUX*, Vol. 1397. 277–292. https://doi.org/10.1007/3-540-69778-0_28

[52] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding High-Performance SAT Solvers with Unsat-Core Predictions. In *SAT*, Vol. 11628. 336–353. https://doi.org/10.1007/978-3-030-24258-9_24

[53] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. 2019. Learning a SAT Solver from Single-Bit Supervision. In *ICLR*. 1–11. https://doi.org/10.48550/arXiv.1802.03685

[54] A. Prasad Sistla and Edmund M. Clarke. 1985. The Complexity of Propositional Linear Temporal Logics. *J. ACM* 32, 3 (1985), 733–749. https://doi.org/10.1145/3828.3837

[55] Richard Socher, Brody Huval, Christopher D. Manning, and Andrew Y. Ng. 2012. Semantic Compositionality through Recursive Matrix-Vector Spaces. In *EMNLP-CoNLL*. 1201–1211.

[56] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic

Compositionality Over a Sentiment Treebank. In *EMNLP*. 1631–1642.

[57] Pashootan Vaezipoor, Andrew C. Li, Rodrigo Toro Icarte, and Sheila A. McIlraith. 2021. LTL2Action: Generalizing LTL Instructions for Multi-Task RL. In *ICML*, Vol. 139. 10497–10508.

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS*. 5998–6008.

[59] Haoyu Wang, Nan Wu, Hang Yang, Cong Hao, and Pan Li. 2022. Unsupervised Learning for Combinatorial Optimization with Principled Objective Relaxation. In *NeurIPS*.

[60] Po-Wei Wang, Priya L. Donti, Bryan Wilder, and J. Zico Kolter. 2019. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *ICML*, Vol. 97. 6545–6554.

[61] Pierre Wolper. 1985. The tableau method for temporal logic: An overview. *Logique et Analyse* (1985), 119–136.

[62] Martin De Wulf, Laurent Doyen, Nicolas Maquet, and Jean-François Raskin. 2008. Antichains: Alternative Algorithms for LTL Satisfiability and Model-Checking. In *TACAS*, Vol. 4963. 63–77. https://doi.org/10.1007/978-3-540-78800-3_6

[63] Martin De Wulf, Laurent Doyen, Nicolas Maquet, and Jean-François Raskin. 2008. Antichains: Alternative Algorithms for LTL Satisfiability and Model-Checking. In *TACAS*, Vol. 4963. 63–77. https://doi.org/10.1007/978-3-540-78800-3_6

[64] Yaqi Xie, Fan Zhou, and Harold Soh. 2021. Embedding Symbolic Temporal Knowledge into Deep Sequential Models. In *ICRA*. 4267–4273. https://doi.org/10.1109/ICRA48506.2021.9561952

[65] Runxin Xu, Fuli Luo, Zhiyuan Zhang, Chuanqi Tan, Baobao Chang, Songfang Huang, and Fei Huang. 2021. Raise a Child in Large Language Model: Towards Effective and Generalizable Fine-tuning. In *EMNLP*. 9514–9528. https://doi.org/10.48550/arXiv.2109.05687

[66] Rongzhen Ye, Tianqu Zhuang, Hai Wan, Jianfeng Du, Weilin Luo, and Pingjia Liang. 2023. A Noise-Tolerant Differentiable Learning Approach for Single Occurrence Regular Expression with Interleaving. In *AAAI*. 4809–4817. https://doi.org/10.1609/aaai.v37i4.25606

[67] Wenjie Zhang, Zeyu Sun, Qihao Zhu, Ge Li, Shaowei Cai, Yingfei Xiong, and Lu Zhang. 2020. NLocalSAT: Boosting Local Search with Solution Prediction. In *IJCAI*. 1177–1183. https://doi.org/10.24963/ijcai.2020/164

Received 2024-04-12; accepted 2024-07-03