NADA: Neural Acceptance-Driven Approximate Specification Mining

WEILIN LUO, Sun Yat-sen University, China
TINGCHEN HAN, Sun Yat-sen University, China
JUNMING QIU, Sun Yat-sen University, China
HAI WAN*, Sun Yat-sen University, China
JIANFENG DU*, Guangdong University of Foreign Studies, China
BO PENG, Sun Yat-sen University, China
GUOHUI XIAO, Southeast University, China
YANAN LIU, Sun Yat-sen University, China

It is hard to mine high-quality finite-state automata (FSAs) only from desired software behaviors, *i.e.*, positive examples, because of a search space explosion and an overgeneralization problem induced by a lack of undesired software behaviors, *i.e.*, negative examples. To tackle the overgeneralization problem, we suggest modeling the problem as searching for *approximate* FSAs from positive and negative examples with *noise*, where the noise originates from synthetic negative examples used to reject overgeneralized results. To obtain an effective search bias in the exploding search space, we bridge FSA acceptance to neural network inference. Our key contribution is to design a neural network whose parameter assignment corresponds to an FSA, and its neural inference process, named after *neural acceptance*, is able to simulate FSA acceptance. The neural acceptance provides a way to quantify how well an FSA fits noisy data efficiently. We propose NADA, a neural acceptance-driven approach, to search for approximate FSAs guided by accepting positive examples and rejecting synthetic negative examples. NADA is based on a proper continuous relaxation of the discrete search space of FSAs and an efficient gradient descent-based search algorithm. Experimental results demonstrate that, compared with state-of-the-art approaches, NADA significantly improves the quality of mined FSAs (on average improves 41.63% F1 score). Besides, NADA is 19.8X faster than the approach mining sub-high-quality FSAs.

CCS Concepts: • **Software and its engineering** → **Dynamic analysis**; *Software testing and debugging*; • **Computing methodologies** → **Neural networks**; • **Theory of computation** → *Modal and temporal logics.*

*Both authors are corresponding authors.

Authors' Contact Information: Weilin Luo, Sun Yat-sen University, School of Computer Science and Engineering, MoE Key Laboratory of Information Technology, Guangzhou, China, luowlin5@mail.sysu.edu.cn; Tingchen Han, Sun Yat-sen University, School of Computer Science and Engineering, Guangzhou, China, hantch@mail2.sysu.edu.cn; Junming Qiu, Sun Yat-sen University, School of Computer Science and Engineering, Guangzhou, China, qiujm9@mail2.sysu.edu.cn; Hai Wan, Sun Yat-sen University, School of Computer Science and Engineering, MoE Key Laboratory of Information Technology, Guangzhou, China, wanhai@mail.sysu.edu.cn; Jianfeng Du, Guangdong University of Foreign Studies, Guangzhou, China, jfdu@gdufs.edu.cn; Bo Peng, Sun Yat-sen University, School of Computer Science and Engineering, Guangzhou, China, pengb8@mail2.sysu.edu.cn; Guohui Xiao, Southeast University, School of Computer Science and Engineering, Nanjing, China, guohui.xiao@seu.edu.cn; Yanan Liu, Sun Yat-sen University, School of Computer Science and Engineering, Guangzhou, China, liuyn56@mail2.sysu.edu.cn.

ACM 2994-970X/2025/7-ARTISSTA079
https://doi.org/10.1145/3728956

Additional Key Words and Phrases: Specification Mining, Finite-state Automata, Deep Learning, Linear Temporal Logic.

ACM Reference Format:
Weilin Luo, Tingchen Han, Junming Qiu, Hai Wan, Jianfeng Du, Bo Peng, Guohui Xiao, and Yanan Liu. 2025. NADA: Neural Acceptance-Driven Approximate Specification Mining. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA079 (July 2025), 23 pages. https://doi.org/10.1145/3728956

1 Introduction

Specifications are able to characterize high-level software behaviors, thereby improving maintainability and reliability. Conversely, the absence of specifications often leads to program comprehension challenges and increases the risk of errors. For example, API misuse has been recognized as a major cause of bugs and vulnerabilities [Nadi et al. 2016]. Besides, developers cannot leverage tools for bug discovery and testing that rely on specifications as input [Miao and Liu 2012]. However, specifications are frequently unavailable in practice due to the high cost and time-intensive nature of their creation, as well as the rapid pace of software evolution. The above contradictions have stimulated an urgent need for automated approaches to mining high-quality specifications. Beyond traditional software, specification mining also plays a crucial role in formalizing properties of data in verified artificial intelligence (AI) software [Seshia et al. 2022].

We examine the specification mining only from positive examples, since negative examples are often difficult to obtain in practice, where the positive examples are desired software behaviors, *e.g.*, execution traces, and the negative examples are undesired software behaviors. We focus on finite-state automata (FSAs) as specifications due to their expressive power in capturing temporal properties, where an FSA accepts positive examples and rejects negative examples. A high-quality FSA requires not only to accept known positive examples, but also to accept unknown positive examples and reject unknown negative examples. However, mining such FSAs only from positive examples presents significant challenges. It confronts a serious issue of search space explosion due to the NP-hard complexity [Gold 1978; Pitt and Warmuth 1993]; moreover, it suffers from an overgeneralization problem, as there is no mechanism to penalize overgeneralized hypotheses.

To obtain high-quality FSAs inexpensively, FSA mining has been extensively studied. Most existing approaches, *e.g.*, [de Caso et al. 2012; Giantamidis et al. 2021; Kang and Lo 2021; Krka et al. 2014; Mariani et al. 2011], employ well-designed heuristics for abstracting states, which we refer to as state abstraction-based approaches. These approaches update the architectures of FSAs by abstracting states, thereby the quality of their mined FSAs depends on the employed heuristics for abstracting states. However, it is hard to design good heuristics and to verify their ubiquitous effectiveness. Alternative constrained optimization-based approaches [Roy et al. 2023; Shvo et al. 2021] formulate FSA mining as an optimization problem with anti-overgeneralization constraints. Although theoretically appealing, these approaches face two key limitations: (1) their effectiveness is bounded by the expressiveness of the encoded anti-overgeneralization properties, and (2) their scalability depends on the performance of the underlying constraint solver.

Recent works have leveraged the representational power of neural networks to advance FSA mining. Some approaches [Cao and Zhang 2020; Le and Lo 2018] have migrated from hand-engineered heuristics toward data-driven ones to improve the quality of heuristics. Besides, a number of works, *e.g.*, [Okudono et al. 2020; Weiss et al. 2024; Zhang et al. 2021] employ a two-stage pipeline: first training a recurrent neural network (RNN) to classify behaviors, then extracting an FSA to mimic the behaviors of the RNN. Above learning-based approaches, however, suffer from the error propagation problem, because they do not directly train neural networks to mine FSAs, but first to learn features of positive examples and then to use the features as heuristics for state

abstraction or just be faithful to the trained RNN, not the original data. Consequently, despite these advances, mining high-quality FSAs remains an open challenge.

In this paper, we suggest modeling the problem as searching for approximate FSAs from positive and negative examples with noise. Noise refers to mislabeled data that contradict ground truths, *i.e.*, false positives/negatives. Reducing noise in negative examples is advantageous for rejecting overgeneralized FSAs. We refer to an example that is highly confident to be a true negative example as a *potential negative example*. We synthesize potential negative examples based on temporal properties that approximate boundaries between positive and negative examples. Its insight is to give priority to generating more specific temporal properties to generalize all positive examples quickly. To alleviate search space explosion, we develop a *FSA encoding* method to parameterize a neural network `FSAccept`, so that its inference process, named after *neural acceptance*, is able to simulate FSA acceptance and that an FSA can be easily interpreted from a parameter assignment of `FSAccept`. Intuitively, the neural acceptance is able to quantify approximately how well an interpreted FSA fits data. In addition, to bridge the gap between neural acceptance and FSA acceptance, we identify *faithful FSA encoding*, a subclass of FSA encoding, which has a one-to-one correspondence to FSAs. It enables a data-driven approach, `NADA`, which guides the mining of high-quality FSAs by directly training `FSAccept` to accept positive examples, reject negative examples and fulfill the faithful FSA encoding. The key insight of `FSAccept` is based on a continuous relaxation of discrete architecture representations, making the search space differentiable. It allows for efficient architecture optimization using a gradient descent algorithm and alleviates the wrong search bias resulting from noise using approximate computation.

We evaluate `NADA` on mining FSAs of 11 datasets, which were commonly used in experiments in related work [Kang and Lo 2021; Krka et al. 2014; Le et al. 2015; Le and Lo 2018]. The baselines with which we compare include state-of-the-art (SOTA) learning-based [Le and Lo 2018; Weiss et al. 2024], state abstraction-based [Kang and Lo 2021], and constrained optimization-based [Roy et al. 2023; Shvo et al. 2021] approaches. Experimental results confirm the effectiveness of `NADA`, demonstrating that it spends less time to achieve higher performance than the state abstraction-based approach, and also achieves higher performance than learning-based and constrained optimization-based approaches. Besides, our results from ablation studies confirm that the faithful FSA encoding and the potential negative examples improve the quality of mined FSAs.

2 Related Work

2.1 FSA Mining

2.1.1 Mining Automata Only from Positive Examples. Although it is impossible to mine FSAs exactly only from positive examples [Gold 1967], there are still many practical approaches to approximately mining FSAs. The main approaches follow a gradual abstraction paradigm. They first build an FSA that only accepts positive examples, *e.g.*, prefix tree acceptor, and then search for more general FSAs via gradually abstracting states. The core is to provide well-designed heuristics for state abstraction to determine whether two strings reach the same state. Biermann and Feldman [1972] proposed *k-tail*, a state abstraction-based approach. Its heuristic is that if the short-term behaviors in two states are the same, then the long-term behaviors in the two states should also be the same, where the short-term behavior of a state is defined by the set of suffix strings of size k or less, and the long-term behavior of a state is defined by the set of suffix strings of size greater than k. After that, various improved k-tail approaches were proposed [Ammons et al. 2002; Kang and Lo 2021; Lo and Khoo 2006b; Lo et al. 2009; Lorenzoli et al. 2008; Mariani et al. 2011; Walkinshaw and Bogdanov 2008] by designing different heuristics for abstracting states. Besides, there are some other abstraction methods [Dallmeier et al. 2010, 2006; de Caso et al. 2012; Krka et al. 2014]. Although the

above approaches work well in some scenarios thanks to well-designed heuristics for abstracting states, they are limited by the quality of heuristics and suffer from non-trivial and time-consuming heuristic designing. Different from the gradual abstraction paradigm, we develop a differentiable quality assessment for candidate FSAs, which enables us to search for high-quality FSAs using a gradient descent-based search algorithm. Some approaches model the problem as a constrained optimization problem, whose core is to construct constraints characterizing high-quality FSAs. Avellaneda and Petrenko [2018] and Roy et al. [2023] used language minimality as a regularizer to guide mining FSAs. However, language minimality does not guarantee good performance in some domains, as demonstrated by our experimental results (Table 2).

2.1.2 Mining Automata from Positive and Negative Examples. Early works [Gold 1978; Trakhtenbrot and Barzdin 1973] also followed the gradual abstraction paradigm. Oncina et al. [1992] and Lang [1992] adopted greedy state merging as a heuristic for state abstraction. Subsequent work further optimized this heuristic. Representative algorithms include three categories: evidence-driven state merging algorithm [Lang et al. 1998], search-based state merging algorithm [Lang 1999; Oliveira and Silva 2001], and encoding algorithm for constraint satisfaction techniques [Angluin 1987; Angluin et al. 2015; Giantamidis et al. 2021; Gold 1967; Heule and Verwer 2010; Oncina and Garcia 1992; Smetsers et al. 2018; Zakirzyanov et al. 2019]. However, none of the above works showed robustness to noisy data. Other works can handle noise [Shvo et al. 2021; Ulyantsev et al. 2015; Xue et al. 2015], which is close to our setting. Shvo et al. [2021] presented a SOTA approach to mining FSAs as interpretable sequence classifiers via discrete optimization. They encoded an FSA mining problem as a mixed integer linear programming (MILP) problem, which is limited by the performance of MILP solvers.

2.2 Learning-Based Mining Approaches

Due to the powerful automatic feature induction of neural networks, some learning-based approaches have been designed. Le and Lo [2018] and Cao and Zhang [2020] trained RNNs to capture sequence features for abstracting states. They did not directly train neural networks to mine FSAs but to learn sequence features, which suffers from the error propagation problem. Some approaches explore the extraction of FSAs from trained RNNs, where the FSA attempts to approximate the RNN's behavior. Early approaches [Angluin 1987; Cechin et al. 2003; Jacobsson 2005; Kolen 1993; Omlin and Giles 1996a,b; Omlin et al. 1996; Schellhammer et al. 1998] are limited by the performance of RNNs. As RNN performance improves, these approaches [Ayache et al. 2018; Grachev et al. 2017; Koul et al. 2019; Ma et al. 2022; Michalenko et al. 2019; Okudono et al. 2020; Suzuki et al. 2021; Wang et al. 2018; Weiss et al. 2018, 2019, 2024; Zhang et al. 2021] still face a critical limitation: they treat RNNs as black-box, requiring complex extraction algorithms to compensate for the substantial performance gap between the RNN and its extracted FSA. In contrast, our approach establishes an explicit and trainable mapping between FSA structures and parameters of neural networks, where an FSA can be extracted just by iterating over the parameters.

Recent advances have demonstrated neural networks' ability to simulate formal language inference [Stogin et al. 2024; Svete et al. 2024]. Parallel developments [Luo et al. 2022; Mordvintsev 2022; Wan et al. 2024; Wang et al. 2023; Yu et al. 2024] have introduced gradient descent-based search approaches that leverage neural networks' semantic alignment with formal languages. These approaches benefit from GPU acceleration, enabling efficient exploration of the solution space. Building on these insights, we propose a novel framework that directly trains neural networks to mine FSAs by simulating FSA acceptance behavior.

2.3 Temporal Properties to Guide Mining

Previous works [Beschastnikh et al. 2015, 2011; Dwyer et al. 1999; Kang and Lo 2021; Le et al. 2015; Lo et al. 2009; Sun et al. 2019; Walkinshaw and Bogdanov 2008] have exploited temporal properties to guide FSA mining. They predominantly relied on template-based formulations with good understandability and conciseness. While we similarly leverage temporal properties, our approach differs from them: we exploit the properties to synthesize potential negative examples, which enable neural networks to reject overgeneralized FSAs. We note that recent advances in AI have made significant progress in learning arbitrary-form LTL formulae [Camacho and McIlraith 2019; Gaglione et al. 2021; Luo et al. 2022; Neider and Gavran 2018; Wan et al. 2024], suggesting that it is promising to enhance FSA mining by more expressive property learning.

3 Background

3.1 Notation of Neural Networks

Throughout this paper, we use lowercase (*resp.* uppercase) bold letters to represent vectors (*resp.* matrices). Let $(\mathbf{v})_i$ denote the i^{th} element of a vector $\mathbf{v}$ and $(\mathbf{A})_{i,j}$ denote the element of a matrix $\mathbf{A}$ at the i^{th} row and the j^{th} column, where i and j start from 1.

3.2 Finite-State Automata

An FSA is a 5-tuple $(\Sigma, S, s_0, \delta, F)$, where Σ is a finite non-empty set of symbols called alphabet, S is a finite non-empty set of states, $s_0 \in S$ is an initial state, $\delta\colon S \times \Sigma \rightarrow 2^S$ is a state-transition function, and $F \subseteq S$ is a set of accepting states. Let $N_{\mathrm{s}}^{\mathrm{m}} \in \mathbb{N}$ be the number of states. An FSA can also be defined as a 5-tuple $(\Sigma, S, \mathbf{I}, \{\mathbf{E}_\sigma | \sigma \in \Sigma\}, \mathbf{T})$, where Σ is an alphabet, S is a set of states, $\mathbf{I} \in \{0,1\}^{N_{\mathrm{s}}^{\mathrm{m}}}$ is an initial vector, $\mathbf{E}_\sigma \in \{0,1\}^{N_{\mathrm{s}}^{\mathrm{m}} \times N_{\mathrm{s}}^{\mathrm{m}}}$ is a transition matrix for each $\sigma \in \Sigma$, where $s_i, s_j \in S$, and $\mathbf{T} \in \{0,1\}^{N_{\mathrm{s}}^{\mathrm{m}}}$ is an accepting vector. $\mathbf{I}$ and $\mathbf{E}_\sigma$ are defined as Equation (1).

$$(\mathbf{I})_i = \begin{cases} 1, & i = 1, \\ 0, & \text{otherwise}, \end{cases} \quad (\mathbf{E}_\sigma)_{i,j} = \begin{cases} 1, & s_{j-1} \in \delta(s_{i-1}, \sigma), \\ 0, & \text{otherwise}, \end{cases} \tag{1}$$

$(\mathbf{I})_i = 1$ means s_{i-1} is an initial state, it is not otherwise. Similarly, $(\mathbf{E}_\sigma)_{i,j}$ and $\mathbf{T}_i$ indicate whether there is a transition between s_{i-1} and s_{j-1} and whether s_{i-1} is an accepting state, respectively.

A string π over Σ is a sequence of symbols and the i^{th} symbol of π is denoted by $\pi[i]$. The size of π is the number of symbols in π, denoted by $|\pi|$. π_i denotes a sub-string of π beginning from the i^{th} symbol, particularly, $\pi_1 = \pi$. Let π be a string over Σ of size n and $\mathcal{A} = (\Sigma, S, s_0, \delta, F)$ an FSA. $\mathcal{A}$ accepts π if there is a sequence of states $\langle q_0, \ldots, q_n \rangle$ such that $q_0 = s_0$, $q_{i+1} \in \delta(q_i, \pi[i+1])$ for $i \in [0, n-1]$, and $q_n \in F$. Given an FSA and a string, FSA acceptance is to recognize whether the FSA accepts the string. Let $\mathcal{A} = (\Sigma, S, \mathbf{I}, \{\mathbf{E}_\sigma | \sigma \in \Sigma\}, \mathbf{T})$ be an FSA and π a string. FSA acceptance can be computed using the forward algorithm [Baum and Petrie 1966] defined as Equation (2).

$$\textsc{Accept}(\mathcal{A}, \pi) = \sigma_{01}\left(\mathbf{I}^T \cdot \left(\prod_{i=1}^{|\pi|} \mathbf{E}_{\pi[i]}\right) \cdot \mathbf{T}\right), \tag{2}$$

where $\sigma_{01}(*) = \min(\max(0, *), 1)$. $\mathcal{A}$ accepts π if $\textsc{Accept}(\mathcal{A}, \pi) = 1$; rejects π otherwise.

High-level software behaviors can be characterized by a ground truth model (GTM). A GTM is an FSA, which only accepts all execution traces. An execution trace is a string over an alphabet, *e.g.*, a set of APIs that are called during software execution. GTMs are often used to evaluate the approaches to FSA mining [Kang and Lo 2021; Le and Lo 2018].

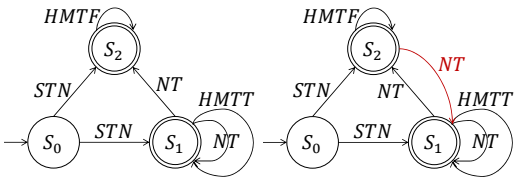
3.3 Linear Temporal Logic

Linear temporal logic (LTL) [Pnueli 1977] has been widely used to characterize temporal properties. The syntax of LTL for a finite set of atomic propositions $\mathbb{P}$, given below, includes logical operators ($\vee$ and $\neg$) and temporal modal operators (next $\bigcirc$ and until $\mathcal{U}$), where $p \in \mathbb{P} \cup \{\top\}$ and φ', φ'' are LTL formulae: $\phi := p \mid \neg\varphi' \mid \varphi' \wedge \varphi'' \mid \bigcirc\varphi \mid \varphi' \,\mathcal{U}\, \varphi''$. Operator or ($\vee$), release ($\mathcal{R}$), eventually ($\Diamond$), always ($\Box$), and weak-until ($\mathcal{W}$) can be defined by the above fundamental operators.

LTL formulae are interpreted over infinite traces of propositional states. An infinite trace is represented in the form $w = w[1], \ldots, (w[k], \ldots, w[n])^{\omega}$, where $w[i] \in 2^{\mathbb{P}}$ is a state at time i and $(w[k], \ldots, w[n])^{\omega}$ is a loop and means that $w[k], \ldots, w[n]$ appears in order and infinitely. w_i denotes a sub-trace of w beginning from the state $w[i]$, particularly, $w_1 = w$. The satisfaction relation $\models$ is defined as follows, where ϕ', ϕ'' are LTL formulae, $i, j, k \in \mathbb{N}$, and $p \in \mathbb{P} \cup \{\top\}$: $w_i \models p$ iff $p \in w[i]$; $w_i \models \neg\phi'$ iff $w_i \not\models \phi'$; $w_i \models \phi' \wedge \phi''$ iff $w_i \models \phi'$ and $w_i \models \phi''$; $w_i \models \bigcirc\phi'$ iff $w_{i+1} \models \phi'$; $w_i \models \phi' \,\mathcal{U}\, \phi''$ iff $\exists i \leq k, w_k \models \phi''$ and $\forall i \leq j < k, w_j \models \phi'$. An LTL formula ϕ implies an LTL formula ψ, denoted by $\phi \rightarrow \psi$, if and only if $\pi \models \psi$ for every trace π such that $\pi \models \phi$.

Since execution traces in specification mining are finite, we use the method [Lemieux et al. 2015] to extend the satisfaction relation to finite traces, which is also used in the work [Kang and Lo 2021]. A finite trace is represented in the form $w = w[1], \ldots, w[n]$, where $w[i] \in 2^{\mathbb{P}}$ is a state at time i. Specifically, we first transform a finite trace into an infinite trace by appending an infinite sequence of terminal symbol ω. Second, let symbols in the alphabet of execution traces be atomic propositions. An execution trace is a special trace where each state has one and only one atomic proposition true. Third, for the operator $\Box$, we transform $\Box\phi$ to $\phi \,\mathcal{U}\, \omega$, where ϕ is an LTL formula. Finally, we check the satisfaction relation according to the definition of $\models$. This extension defines exactly the infinite trace that we are examining, thereby obviating the need for more sophisticated adaptations of LTL for finite traces [Bauer et al. 2007; Lemieux et al. 2015]. Unless stated otherwise, traces are finite in this paper. A string or execution trace corresponds to a trace where one and only one atomic proposition is true at each time.

LTL formulae are able to characterize temporal properties of an FSA, that is, to specify order constraints between symbols. Informally, if an FSA satisfies an LTL formula, then all strings accepted by the FSA satisfy the LTL formula. For example, Kang and Lo [2021] guided a test generator to generate strings that are uncommon, but positive via LTL formulae; Le et al. [2015] used a model checker to check LTL formulae satisfied by an FSA to fission the FSA as a set of LTL formulae.



(a) The GTM. (b) A mined FSA.

Fig. 1. Some FSAs of StringTokenizer. The FSAs include interactions among four APIs: *STN*, *HMTT*, *HMTF*, and *NT*, where '*STN*' is StringTokenizer(), '*HMTT*' is hasMoreTokens() = true, '*HMTF*' is hasMoreTokens() = false, and '*NT*' is nextToken().

4 Motivating Example

Example 1 illustrates some FSAs of the library class StringTokenizer. In programming, a library class refers to a pre-written class that is part of a software library, providing reusable functionality to simplify development.

Example 1. *The GTM* $\mathcal{A} = (\Sigma, S, \mathbf{I}, \{\mathbf{E}_\sigma | \sigma \in \Sigma\}, \mathbf{T})$ *of StringTokenizer is shown in Fig. 1(a)*[1]*, where* s_0 *is an initial state and* s_1*,* s_2 *are accepting states. Suppose a set of positive examples* Π^+ *includes:*

[1]We use the refined version of the work [Le and Lo 2018], so there are some differences from the GTM shown in the work [Le et al. 2015].

$\langle STN, NT\rangle$, $\langle STN, HMTT, NT\rangle$, $\langle STN, HMTF\rangle$, *and* $\langle STN, NT, NT\rangle$. $\pi = \langle STN, NT\rangle$ *is a positive example because* $\textsc{Accept}(\mathcal{A}, \pi) = \sigma_{01}\left(\mathbf{I}^T \cdot \mathbf{E}_{STN} \cdot \mathbf{E}_{NT} \cdot \mathbf{T}\right) =$

$$\sigma_{01}\left(\begin{bmatrix}1\\0\\0\end{bmatrix}^T \cdot \begin{bmatrix}0&1&1\\0&0&0\\0&0&0\end{bmatrix} \cdot \begin{bmatrix}0&0&0\\0&1&1\\0&0&0\end{bmatrix} \cdot \begin{bmatrix}0\\1\\1\end{bmatrix}\right) = 1.$$

Due to the lack of negative examples, it is difficult to reject over-generalized FSAs for existing approaches. For example, they may mine an FSA form Π^+ shown in Fig. 1(b) because the FSA accepts all positive examples in Π^+. We suggest to synthesize potential negative examples to reject over-generalized FSAs. Example 2 demonstrates the synthesizing process.

Example 2 (Example 1 continued). *We first generate a set of LTL formulae* $\Phi = \{\square(HMTF \rightarrow \bigcirc\square\neg NT)\}$ *from* Π^+ *because for all* $\pi \in \Pi^+$, $\pi \models \square(HMTF \rightarrow \bigcirc\square\neg NT)$. *Then, we synthesize a set of potential negative examples* $\Pi^-_{\mathbf{u}} = \{\langle STN, HMTF, NT\rangle\}$ *because* $\langle STN, HMTF, NT\rangle \not\models \square(HMTF \rightarrow \bigcirc\square\neg NT)$ *and* $\langle STN, HMTF, NT\rangle$ *is close to* $\langle STN, HMTT, NT\rangle$, *i.e., they only differ in one symbol.*

Driven by the neural acceptance, we directly train `FSAccept` to accept positive examples and reject potential negative examples. Example 3 demonstrates the high-quality search bias.

Example 3 (Example 2 continued). *We define that* $\{\mathbf{W}_\sigma | \sigma \in \Sigma\}$ *and* $\mathbf{F}$ *are trainable parameters of* `FSAccept` *and their assignment corresponds to an FSA encoding* $\left(\left\{\tilde{\mathbf{W}}_\sigma | \sigma \in \Sigma\right\}, \tilde{\mathbf{F}}\right)$. *Suppose an FSA encoding is as follows (it is partial for brevity):*

$$\tilde{\mathbf{W}}_{STN} = \begin{bmatrix}0.01&0.99&0.96\\0.02&0.08&0.11\\0.07&0.04&0.01\end{bmatrix}, \quad \tilde{\mathbf{W}}_{HMTF} = \begin{bmatrix}0.05&0.09&0.01\\0.12&0.05&0.01\\0.02&0.01&0.95\end{bmatrix}, \quad \tilde{\mathbf{W}}_{NT} = \begin{bmatrix}0.01&0.48&0.49\\0.02&0.96&0.89\\0.01&0.03&0.02\end{bmatrix}$$

and $\tilde{\mathbf{F}} = [0.01\ 0.99\ 0.95]^T$. *An FSA encoding models a set of FSAs, where* $\left\{\tilde{\mathbf{W}}_\sigma | \sigma \in \Sigma\right\}$ *and* $\tilde{\mathbf{F}}$ *indicate the probability of transition and that of accepting state respectively. Given a positive example* $\langle STN, NT\rangle$, *the neural acceptance computes the probability that* $\langle STN, NT\rangle$ *is a positive example:*

$$\sigma^d_{01}(\sigma^d_{01}(\sigma^d_{01}(\mathbf{I}^T\tilde{\mathbf{W}}_{STN})\tilde{\mathbf{W}}_{NT})\tilde{\mathbf{F}}), \tag{3}$$

where σ^d_{01} *is a differentiable approximation of* σ_{01}. *We train* `FSAccept` *by minimizing the loss function defined by Equation (3), where the gradient characterizes the bias of the FSAs corresponding to the parameter toward accepting* $\langle STN, NT\rangle$. *Similarly, for the potential negative examples* $\langle STN, HMTF, NT\rangle$, *the gradient also characterizes the bias of that toward rejecting* $\langle STN, HMTF, NT\rangle$.

5 Neural Acceptance-Driven Approximation

In this section, we first provide an overview of `NADA`, then introduce the synthesis of potential negative examples and the neural acceptance-driven mining.

5.1 Overview of `NADA`

We formulate mining FSAs as searching for approximate FSAs from a set of positive examples Π^+ and a set of potential negative examples $\Pi^-_{\mathbf{u}}$ by updating an FSA $\mathcal{A}$:

$$\min_{\mathcal{A}} \sum_{\pi \in \Pi^-_{\mathbf{u}}} \textsc{Accept}(\mathcal{A}, \pi) - \sum_{\pi \in \Pi^+} \textsc{Accept}(\mathcal{A}, \pi) \tag{4}$$

We propose a data-driven approach, NADA, to deal with this problem. Specifically, we first generate a set of LTL formulae to guide synthesizing potential negative examples (line 1 of Alg. 1). Then, we synthesize a set of potential negative examples Π^-_{val} (line 2 of Alg. 1) which is used to evaluate the quality of the learned FSA (line 8 of Alg. 1). We also synthesize a set of potential negative examples Π^-_{tra} to form a training dataset (lines 3-4 of Alg. 1). Final, we iteratively search for FSAs based on the gradient descent (line 7 of Alg. 1) until reaching the maximum epoch, a hyper-parameter.

Algorithm 1: NADA

Input : A set of positive examples Π^+.
Output : An FSA $\mathcal{A}^*$.

1 $\Phi \leftarrow$ GenerateLTL(Π^+)
2 $\Pi^-_{val} \leftarrow$ SynthesizePNE(Π^+,Φ)
3 $\Pi^-_{tra} \leftarrow$ SynthesizePNE(Π^+,Φ)
4 $D \leftarrow \{(x, y) \,|x \in \Pi^+, y = 1\} \cup \{(x, y) \,|x \in \Pi^-_{tra}, y = 0\}$
5 $loss^* \leftarrow |\left\{\Pi^+ \cup \Pi^-_{val}\right\}|$, $\mathcal{A}^* \leftarrow NULL$
6 **for** *reach the maximum epoch* **do**
7 $\quad \mathcal{A} \leftarrow$ SearchFSAbyDradient(D)
8 $\quad loss \leftarrow \sum_{\pi \in \Pi^-_{val}}$ Accept($\mathcal{A}, \pi$) − $\sum_{\pi \in \Pi^+}$ Accept($\mathcal{A}, \pi$)
9 $\quad$ **if** $loss < loss^*$ **then**
10 $\quad\quad loss^* \leftarrow loss$, $\mathcal{A}^* \leftarrow \mathcal{A}$
11 **return** $\mathcal{A}^*$

5.2 Synthesizing Potential Negative Examples Based on LTL

Given the high expense associated with manually procuring negative examples, we propose to automatically synthesize negative examples as a viable alternative. Synthesized negative examples may be unknown positive examples, which we call noise. Our objective is to synthesize negative examples containing a small amount of noise, as the noise-free negative examples are instrumental in accurately rejecting overgeneralized FSAs. The previous works [Beschastnikh et al. 2015, 2011; Dwyer et al. 1999; Kang and Lo 2021; Le et al. 2015; Lo et al. 2009; Sun et al. 2019; Walkinshaw and Bogdanov 2008] have verified that introducing temporal properties, represented by LTL formulae, is able to prevent overgeneralizing. Its intuition is to refine FSAs by making FSAs fulfill temporal properties. Inspired by this intuition, we harness the temporal properties that FSAs are expected to satisfy to guide the synthesis of potential negative examples with reduced noise, as demonstrated in Algorithm 3.

Algorithm 2: GenerateLTL

Input : Positive examples Π^+.
Output : A set of temporal properties Φ.

1 $\Phi \leftarrow \emptyset$
2 **foreach** *symbol pair p and q* **do**
3 $\quad \Phi_h \leftarrow \{\square(p \rightarrow \bigcirc q), \square(p \rightarrow \bigcirc\square\neg q), \Diamond(p) \rightarrow (\neg p \,\mathcal{U}\, (q \wedge \bigcirc p))\}$
4 $\quad \Phi_l \leftarrow \{\square(p \rightarrow \bigcirc\Diamond q), \square(p \rightarrow \bigcirc\neg q), \neg p \,\mathcal{W}\, q\}$
5 $\quad$ define a mapping $\triangle$ from Φ_h to Φ_l such that $\triangle(\square(p \rightarrow \bigcirc q)) := \square(p \rightarrow \bigcirc\Diamond q)$, $\triangle(\square(p \rightarrow \bigcirc\square\neg q)) := \square(p \rightarrow \bigcirc\neg q)$, and $\triangle(\Diamond(p) \rightarrow (\neg p \,\mathcal{U}\, (q \wedge \bigcirc p))) := \neg p \,\mathcal{W}\, q$
6 $\quad$ **foreach** $\phi \in \Phi_h$ **do**
7 $\quad\quad$ **if** $\pi \models \phi$ *for all* $\pi \in \Pi^+$ **then**
8 $\quad\quad\quad \Phi \leftarrow \Phi \cup \{\phi\}$, $\Phi_l \leftarrow \Phi_l \setminus \{\triangle(\phi)\}$
9 $\quad$ **foreach** $\phi \in \Phi_l$ **do**
10 $\quad\quad$ **if** $\pi \models \phi$ *for all* $\pi \in \Pi^+$ **then**
11 $\quad\quad\quad \Phi \leftarrow \Phi \cup \{\phi\}$
12 **return** Φ

Before that, we generate LTL formulae to characterize the temporal properties of positive examples, denoted by Φ. For efficiency, we only consider six LTL formula templates with two symbols [Beschastnikh et al. 2015; Dwyer et al. 1999; Le et al. 2015], described as follows, where p and q are symbols: $\square(p \rightarrow \bigcirc\Diamond q)$: p is always followed by q; $\square(p \rightarrow \bigcirc\square\neg q)$: p is never followed by q; $\neg p \,\mathcal{W}\, q$: p is always preceded by q; $\square(p \rightarrow \bigcirc q)$: p is always immediately followed by q; $\square(p \rightarrow \bigcirc\neg q)$: p is never immediately followed by q; $\Diamond(p) \rightarrow (\neg p \,\mathcal{U}\, (q \wedge \bigcirc p))$: p is always immediately preceded by q. Le et al. [2015] proposed a straightforward method to generate LTL formulae constrained by the above templates. Specifically, given an alphabet and a set of positive examples over the alphabet, they first enumerated all possible symbol pairs to instantiate six templates. Then,

they constructed Φ by selecting all the LTL formulae that all positive examples satisfy. Although limiting the number of variables in a template, enumerating all template-constrained LTL formulae is still time-consuming.

For efficiency, we propose a *specificity-first* generation method shown in Algorithm 2. We traverse all formulae in a partial order to filter out more general formulae. Its insight is to generate more specific temporal properties of FSA. We define the partial order via the implication relation of LTL, *i.e.*, given two LTL formulae ϕ and ψ, if $\phi \rightarrow \psi$, then ϕ is ordered before ψ. Specifically, we prioritize evaluations of more specific formulae (lines 6-8 of Alg. 2). If they can already satisfy all positive examples, evaluations of more general formulae will be skipped (line 8 of Alg. 2). It can generate a smaller set of LTL formulae and ensure that the sampling space of negative examples is consistent with the method [Le et al. 2015].

By utilizing Φ, we synthesize a corresponding potential negative example for every positive example to maintain a balance between positive and negative examples (lines 2-5 of Alg. 3), which is advantageous for the gradient descent-based search process. A straightforward method involves iteratively and randomly substituting a symbol within each positive example until it breaches a temporal property. However, in the worst case, the iteration may enter an infinite loop, preventing termination.

We impose constraints on temporal properties through specific mutation conditions. Let ϕ be an LTL formula and π a string. We define mutation conditions as that if ϕ is $\square(p \rightarrow \bigcirc\diamond q)$, $\square(p \rightarrow \bigcirc\square\neg q)$, $\neg p \,\mathcal{W}\, q$, $\square(p \rightarrow \bigcirc q)$, $\square(p \rightarrow \bigcirc\neg q)$, or $\diamond(p) \rightarrow (\neg p \,\mathcal{U}\, (q \wedge \bigcirc p))$, then p must be in π. Given a string π, we randomly select a temporal property ϕ fulfilling the mutation conditions and fed π and ϕ to Algorithm 4 as inputs. Thanks to the mutation conditions, Algorithm 4 runs in linear time and guarantees that its outcome does not satisfy ϕ. Besides, Algorithm 4 is able to synthesize a example that is close to π, *i.e.*, they differ only in a few symbols, which is beneficial for capturing boundaries of positive and negative examples.

Algorithm 3: SynthesizePNE

Input : Positive examples Π^+ and a set of LTL formulae Φ.
Output : Potential negative examples $\Pi^-_{\mathbf{u}}$.

1 $\Pi_{\mathbf{u}} \leftarrow \emptyset$
2 **foreach** $\pi \in \Pi^+$ **do**
3 randomly obtain an LTL formula $\phi \in \Phi$ such that $\pi \models \phi$ and ϕ fulfills the mutation conditions
4 mutate π to π' via Algorithm 4
5 $\Pi_{\mathbf{u}} \leftarrow \Pi_{\mathbf{u}} \cup \{\pi'\}$
6 **return** $\Pi^-_{\mathbf{u}}$

Algorithm 4: Mutate positive examples.

Input : A positive example π and a temporal property ϕ.
Output : A potential negative example π'.

1 **if** $\phi = \square(p \rightarrow \bigcirc\diamond q)$ **then**
2 $\pi' \leftarrow$ randomly select a $p \in \pi$, and remove all q following p
3 **else if** $\phi = \square(p \rightarrow \bigcirc\square\neg q)$ **then**
4 $\pi' \leftarrow$ randomly select a $p \in \pi$, and insert q at a random position following p
5 **else if** $\phi = \neg p \,\mathcal{W}\, q$ **then**
6 $\pi' \leftarrow$ select the first $p \in \pi$ and the $q \in \pi$ preceding p, and exchange q and p
7 **else if** $\phi = \square(p \rightarrow \bigcirc q)$ **then**
8 $\pi' \leftarrow$ randomly select a $p \in \pi$, and remove the q immediately following p
9 **else if** $\phi = \square(p \rightarrow \bigcirc\neg q)$ **then**
10 $\pi' \leftarrow$ randomly select a $p \in \pi$, and insert a q immediately following p
11 **else if** $\phi = \diamond(p) \rightarrow (\neg p \,\mathcal{U}\, (q \wedge \bigcirc p))$ **then**
12 $\pi' \leftarrow$ select the first $p \in \pi$ and the $q \in \pi$ immediately preceding p, and exchange q and p
13 **return** π'

Although potential negative examples may be unknown positive examples, our method is reasonable. Potential negative examples do not fulfill temporal properties satisfied by the positive examples, making them have a high probability of being true. We empirically confirm that in Table 2. Besides, approximate computation defined in Section 5.3 can deal with noise.

5.3 Searching FSAs Based on Gradient Descent

We design a neural network FSAccept, which is able to interpret FSAs from a parameter assignment, to simulate FSA acceptance. Its insight is to model FSA mining formalized in discrete domain as parameter learning defined in continuous domain. We are able to search for high-quality FSAs with the help of neural network training based on gradient descent and cope with noise via the approximate computation of neural networks.

5.3.1 Bridging FSA Acceptance to Neural Network Inference. We define the parameters of FSAccept as Definition 1. $\{\mathbf{W}_\sigma|\sigma\in\Sigma\}$ models a distribution of state transition, $\mathbf{F}$ models a distribution of accepting states, and N_s^m, a hyper-parameter, represents the number of states.

Definition 1. *Let Σ be an alphabet and $N_s^m \in \mathbb{N}$. The trainable parameters of* FSAccept *is a tuple $\theta = (\{\mathbf{W}_\sigma|\sigma\in\Sigma\}, \mathbf{F})$, where $\mathbf{W}_\sigma \in \mathbb{R}^{N_s^m\times N_s^m}$ and $\mathbf{F}\in\mathbb{R}^{N_s^m}$.*

To establish a relationship between parameters of FSAccept and an FSA, we confine that every parameter in θ is assigned a value between 0 and 1. We call a parameter assignment with this restriction an *FSA encoding* defined as Definition 2. Intuitively, an FSA encoding is able to represent an FSA. For all $i,j\in[1,N_s^m]$ and $\sigma\in\Sigma$, $(\tilde{\mathbf{W}}_\sigma)_{i,j}$ indicates the probability of transition from state s_{i-1} to state s_{j-1} by reading symbol σ and $(\tilde{\mathbf{F}})_i$ indicates the probability that state s_{i-1} is an accepting state. Therefore, an FSA encoding corresponds to a *soften FSA* $\mathcal{A}_{\tilde\theta} = \left(\Sigma, S, \mathbf{I}, \left\{\tilde{\mathbf{W}}_\sigma|\sigma\in\Sigma\right\}, \tilde{\mathbf{F}}\right)$, where $S=\left\{s_0,\ldots,s_{N_s^m-1}\right\}$ and $\mathbf{I}\in\{0,1\}^{N_s^m}$ is the initial vector defined as Equation (1). In particular, if the value in $\left\{\tilde{\mathbf{W}}_\sigma|\sigma\in\Sigma\right\}$ and $\tilde{\mathbf{F}}$ is 0 or 1, then $\mathcal{A}_{\tilde\theta}$ is an FSA.

Definition 2. *Let Σ be an alphabet and $N_s^m\in\mathbb{N}$.* FSA encoding *is a tuple $\tilde\theta = \left(\left\{\tilde{\mathbf{W}}_\sigma|\sigma\in\Sigma\right\}, \tilde{\mathbf{F}}\right)$, where $\tilde{\mathbf{W}}_\sigma \in \mathbb{R}_{[0,1]}^{N_s^m\times N_s^m}$, $\tilde{\mathbf{F}}\in\mathbb{R}_{[0,1]}^{N_s^m}$, and $\mathbb{R}_{[0,1]}$ denotes the real value range from 0 to 1.*

Algorithm 5: DifAccept

```
Input  : A string π.
Output : Whether A_θ̃ accepts π.
1 x ← I^T
2 foreach i in [1, |π|] do
3     x ← σ01(x · W̃_π[i])
4 return σ01(x · F̃)
```

Algorithm 6: Encode

```
Input  : An FSA A = (Σ, S, I, {E_σ|σ∈Σ}, T) and an integer N_s^m ≥ |S|.
Output : Assigned parameters θ of FSAccept.
1 (W_σ)_{i,j} ← −ϵ for all i, j ∈ [1, N_s^m] and σ ∈ Σ
2 (F)_i ← −ϵ for all i ∈ [1, N_s^m]
3 foreach σ ∈ Σ and i, j ∈ [1, N_s^m] do
4     if (E_σ)_{i,j} = 1 then
5         (W_σ)_{i,j} ← ϵ
6     if (T)_i = 1 then
7         (F)_i ← ϵ
8 return ({W_σ|σ∈Σ}, F)
```

An arbitrary parameter assignment of FSAccept can be converted to an FSA encoding, as follows.

$$(\tilde{\mathbf{W}}_\sigma)_{i,j} = \begin{cases} 0, & (\mathbf{W}_\sigma)_{i,j}\le -\epsilon,\\ 1, & (\mathbf{W}_\sigma)_{i,j}\ge\epsilon,\\ \mathrm{sigmoid}((\mathbf{W}_\sigma)_{i,j}), & otherwise,\end{cases} \quad (\tilde{\mathbf{F}})_i = \begin{cases} 0, & (\mathbf{F})_i\le -\epsilon,\\ 1, & (\mathbf{F})_i\ge\epsilon,\\ \mathrm{sigmoid}((\mathbf{F})_i), & otherwise,\end{cases} \tag{5}$$

where ϵ is a large positive integer (we set $\epsilon = 10^3$) and sigmoid is the sigmoid function.

Given a string π, we define the *neural acceptance* as Algorithm 5 to approximate the FSA acceptance. $\mathbf{x}\in\mathbb{R}^{N_s^m}$ is the state vector and initialized to the initial vector of $\mathcal{A}_{\tilde\theta}$ (line 1 of Alg. 5). For each iteration i, each $(\mathbf{x})_j$ for $j\in[1,N_s^m]$ indicates whether s_{j-1} is reachable under the string $\pi[1],\ldots,\pi[i]$ (line 3 of Alg. 5). Finally, we check whether the accepting states are reachable (line 4 of Alg. 5). By DifAccept$(\pi|\theta)$ we denote whether $\mathcal{A}_{\tilde\theta}$ accepts π. We set that $\mathcal{A}_{\tilde\theta}$ accepts π if DifAccept$(\pi|\theta)\ge 0.5$, $\mathcal{A}_{\tilde\theta}$ rejects π otherwise. Given an FSA $\mathcal{A}$, we introduce Algorithm 6 to encode $\mathcal{A}$ into FSAccept of an equal or greater number of states.

Theorem 1. *Let $\mathcal{A}$ be an FSA with N_s^m states. For any string π,* $\textsc{Accept}(\mathcal{A}, \pi) = 1$ *if and only if* $\textsc{DifAccept}(\pi|\textsc{Encode}(\mathcal{A}, N_s^m)) = 1$.

Theorem 1 can be straightforwardly proved by induction on the size of π. Intuitively, it shows that the neural acceptance is able to simulate FSA acceptance. Therefore, given a set of positive and negative examples, computing DifAccept for each example can provide a high-quality quantification of how well the FSA corresponding to the FSA encoding fits the set.

Since Equation (5) is not continuous and $\sigma_{01}(*)$ is defined in the discrete domain, we perform continuous relaxation on them via their differentiably approximate functions so that the gradient can be propagated. Specifically, we exploit the sigmoid function to differentiable approximate Equation (5), *i.e.*, $(\tilde{\mathbf{W}}_\sigma)_{i,j} = \text{sigmoid}((\mathbf{W}_\sigma)_{i,j})$ for all $i, j \in [1, N_s^m]$ and $\sigma \in \Sigma$; $(\tilde{\mathbf{F}})_i = \text{sigmoid}((\mathbf{F})_i)$ for all $i \in [1, N_s^m]$. The larger ϵ is, the closer the approximation function is to Equation (5). We exploit a variant of leaky ReLU (Equation (6)) as the differentiable approximation of the function $\min(\max(0, *), 1)$, where $\alpha \in \mathbb{R}$ is a hyper-parameter.

$$\sigma_{01}^d(x) = \begin{cases} \alpha x, & x < 0, \\ x, & 0 \le x \le 1, \\ \alpha x + 1 - \alpha, & x > 1. \end{cases} \tag{6}$$

With the above differentiable approximation, the neural acceptance is a differentiable computation. It enables a new idea to search FSAs: based on the FSA encoding, we consider parameter assignments of `FSAccept` as candidate FSAs; we first train `FSAccept` guided by the loss function defined by the neural acceptance and then interpret an FSA from the trained `FSAccept`.

Given training data D, we train `FSAccept` via minimizing the joint loss function:

$$\zeta = \zeta_g + \beta\zeta_{f1} + \gamma\zeta_{f2},\ \zeta_g = \frac{1}{|D|}\sum_{(x,y)\in D}(\textsc{DifAccept}(x|\theta) - y)^2, \tag{7}$$

where $\beta, \gamma \in \mathbb{R}$ is hyper-parameters, ζ_g is to approximate FSA acceptance, and ζ_{f1} and ζ_{f2} are the regularization to achieve higher performance, defined in Section 5.3.3. SearchFSAbyDradient in Algorithm 1 is a one-epoch training process where the data is split into batches, and candidate FSAs are updated batch by batch to fit the data better.

5.3.2 Interpreting FSAs. We design `FSAInter` to recommend an FSA based on the interpretation possibility (IP) defined as Definition 3. An IP measures a local definition of a candidate FSA. For example, if $\text{sim}((\mathbf{W}_\sigma)_{i,j})$ is close to 1, then a candidate FSA fulfills $s_{j-1} \in \delta(s_{i-1}, \sigma)$, where σ is a symbol, s_{i-1} and s_{j-1} are states, and δ is state-transition function. The greater an IP is, the more likely a local definition holds. The total IP is the product of the IPs of all parameters.

Definition 3. *Let Σ be an alphabet, $N_s^m \in \mathbb{N}$, and $(\{\mathbf{W}_\sigma|\sigma \in \Sigma\}, \mathbf{F})$ trained parameters of* `FSAccept`. *The* interpretation possibility (PI) *of $r \in \left\{(\mathbf{W}_\sigma)_{i,j}, (\mathbf{F})_i | \sigma \in \Sigma, i, j \in [1, N_s^m]\right\}$ is defined as* $\text{sim}(r) = \text{sigmoid}(r)$.

Algorithm 7 shows the process of interpreting. Overall, we update a tuple of an FSA and its score based on IPs by traversing trained parameters (lines 4-15 of Alg. 7). C is used to record a locally defined FSA with its IP. C is initialized as an FSA that does not have state transition and accepting states (lines 1-3 of Alg. 7). Each r produces different local definitions of FSAs: if $r \in \left\{(\mathbf{W}_\sigma)_{i,j}|\sigma \in \Sigma, i, j \in [1, N_s^m]\right\}$, then $s_{j-1} \in \delta(s_{i-1}, \sigma)$ or $s_{j-1} \notin \delta(s_{i-1}, \sigma)$; if $r \in \left\{(\mathbf{F})_i | i \in [1, N_s^m]\right\}$, then $s_{i-1} \in F$ or $s_{i-1} \notin F$. We select the locally defined FSA with the best IP to update C (lines 6-15 of Alg. 7). Example 4 illustrates Algorithm 7.

Example 4 (Example 3 continued). *Suppose trained parameters are shown in Example 3. Since* $\text{sigmoid}((\mathbf{W}_{STN})_{1,2}) > 1 - \text{sigmoid}((\mathbf{W}_{STN})_{1,2})$, $\delta(s_0, STN) = \{s_1\}$ *(lines 6-15 of Alg. 7). Similarly, we obtain* $\delta(s_0, STN) = \{s_1, s_2\}$, $\delta(s_2, HMTF) = \{s_2\}$, $\delta(s_1, NT) = \{s_1, s_2\}$, *and* $F = \{s_1, s_2\}$.

5.3.3 Faithful FSA Encoding. It is hard to interpret a unique FSA from an FSA encoding resulting from some ambiguity. For example, in Example 4, $(\tilde{\mathbf{W}_{NT}})_{1,2} = 0.48$, which cannot confirm $s_1 \in \delta(s_0, NT)$ or $s_1 \notin \delta(s_0, NT)$ because $(\tilde{\mathbf{W}_{NT}})_{1,2}$ has no obvious tendency. The ambiguity leads to a huge performance gap between the neural acceptance and the FSA acceptance. For example, in Example 4, the string $\langle NT \rangle$ can be accepted because $\textsc{DifAccept}(\langle NT \rangle | \theta) = 0.94$ while it is rejected by the interpreted FSA. Besides, if neural networks are only trained for higher performance, *i.e.*, minimizing ζ_g, it is easy to suffer from overfitting problems for neural networks empirically. Errors arise due to undesirable and uninterpretable dependencies and associations neural networks learn to store in some high-dimensional hidden state [Cai et al. 2017]. To alleviate the above issues, we design a regularization to guide training a *faithful FSA encoding* (Definition 4), which is a subclass of parameter assignments such that a soften FSA of `FSAccept` and an FSA correspond one to one. Intuitively, the faithful condition (1) guarantees that a soften FSA of `FSAccept` has at least one accepting state; the faithful condition (2) and (3) clear the state-transition function and accepting states, respectively.

Algorithm 7: `FSAInter`

Input : An alphabet Σ, the number of states $N_{\mathbf{s}}^{\mathbf{m}}$, trained parameters $\theta = (\{\mathbf{W}_\sigma | \sigma \in \Sigma\}, \mathbf{F})$.
Output : An FSA $\mathcal{A}$.

1 $S \leftarrow \{s_i | i \in [0, N_{\mathbf{s}}^{\mathbf{m}})\}$
2 $\mathcal{A} \leftarrow (\Sigma, S, s_0, \delta, F)$, where $\delta(s_i, \sigma) = \emptyset$ for all $s_i \in S$ and $\sigma \in \Sigma$ and $F = \emptyset$
3 $C \leftarrow (\mathcal{A}, 1)$
4 **foreach** $r \in \{(\mathbf{W}_\sigma)_{i,j}, (\mathbf{F})_i | \sigma \in \Sigma, i, j \in [1, N_{\mathbf{s}}^{\mathbf{m}}]\}$ **do**
5 $(\mathcal{B}_0, score_0), (\mathcal{B}_1, score_1) \leftarrow C$
6 $score_0 \leftarrow score_0 \times (1 - \text{sim}(r))$
7 $score_1 \leftarrow score_1 \times \text{sim}(r)$
8 **if** r *is* $(\mathbf{W}_\sigma)_{i,j}$ **then**
9 update $\delta(s_{i-1}, \sigma)$ of $\mathcal{B}_1$ as $\delta(s_{i-1}, \sigma) \cup \{s_{j-1}\}$
10 **else**
11 update F of $\mathcal{B}_1$ as $F \cup \{s_{j-1}\}$
12 **if** $score_0 > score_1$ **then**
13 $C \leftarrow (\mathcal{B}_0, score_0)$
14 **else**
15 $C \leftarrow (\mathcal{B}_1, score_1)$
16 $(\mathcal{A}, score) \leftarrow C$
17 **return** $\mathcal{A}$

Definition 4. *Let* Σ *be an alphabet and* $N_{\mathbf{s}}^{\mathbf{m}} \in \mathbb{N}$. *An FSA encoding* $\tilde{\theta} = \left(\{\tilde{\mathbf{W}}_\sigma | \sigma \in \Sigma\}, \tilde{\mathbf{F}}\right)$ *is* faithful *if it fulfills the following faithful conditions.*

(1) *There is at least one* $i \in [1, N_{\mathbf{s}}^{\mathbf{m}}]$ *such that* $(\tilde{\mathbf{F}})_i = 1$.
(2) *For every* $\sigma \in \Sigma$, $i, j \in [1, N_{\mathbf{s}}^{\mathbf{m}}]$, $(\tilde{\mathbf{W}}_\sigma)_{i,j} = 0$ *or* 1.
(3) *For every* $i \in [1, N_{\mathbf{s}}^{\mathbf{m}}]$, $(\tilde{\mathbf{F}})_i = 0$ *or* 1.

To make a trained parameter assignment close to a faithful FSA encoding, we design regularization terms ζ_{f1} and ζ_{f2} to guide training. $\zeta_{f1} = 1 - \max(\tilde{\mathbf{F}})$, where $\max(\tilde{\mathbf{F}})$ returns the maximum value in $\{(\tilde{\mathbf{F}})_i | i \in [1, N_{\mathbf{s}}^{\mathbf{m}}]\}$, which has a differentiable approximate function. ζ_{f1} leads to at least one accepting state, which makes a parameter assignment approximately fulfill the faithful condition (1). We define ζ_{f2} as Equation (8), where $\text{ReLU}(*)$ is the ReLU function. ζ_{f2} makes all values in $\{\tilde{\mathbf{W}}_\sigma | \sigma \in \Sigma\}$ and $\tilde{\mathbf{F}}$ close to 0 or 1, which makes a parameter assignment approximately fulfill the

faithful condition (2) and (3).

$$\begin{aligned}\zeta_{f2} =& \frac{1}{|\Sigma| N_{\mathbf{s}}^{\mathbf{m}} N_{\mathbf{s}}^{\mathbf{m}}} \sum_{\sigma \in \Sigma} \sum_{i=1}^{N_{\mathbf{s}}^{\mathbf{m}}} \sum_{j=1}^{N_{\mathbf{s}}^{\mathbf{m}}} (-\mathrm{ReLU}((\tilde{\mathbf{W}}_\sigma)_{i,j} - 0.5) - \mathrm{ReLU}(0.5 - (\tilde{\mathbf{W}}_\sigma)_{i,j})) \\ &+ \frac{1}{|\Sigma| N_{\mathbf{s}}^{\mathbf{m}}} \sum_{\sigma \in \Sigma} \sum_{i=1}^{N_{\mathbf{s}}^{\mathbf{m}}} (-\mathrm{ReLU}((\tilde{\mathbf{F}})_i - 0.5) - \mathrm{ReLU}(0.5 - (\tilde{\mathbf{F}})_i)).\end{aligned} \tag{8}$$

6 Experimental Results

In this section, we conducted experiments to answer four research questions.

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

RQ 2. *Is every module of* NADA *designed reasonably?*

RQ 3. *How does noise impact the FSA mining?*

RQ 4. *How strong is the robustness to hyper-parameters?*

6.1 Baseline

We compared against 5 baselines. They cover the learning-based [Le and Lo 2018; Weiss et al. 2024], state abstraction-based [Kang and Lo 2021], and constrained optimization-based [Roy et al. 2023; Shvo et al. 2021] approaches. Since we reduce the problem to mining FSAs from positive and negative examples, our baselines also cover two categories of approaches that mine FSAs from only positive examples and from both positive and negative examples.

DSM. Le and Lo [2018] first trained an RNN only from positive examples to produce features characterizing each state, and then used the learned features to abstract states. Their results showed that DSM significantly supered previous approaches: k-tails [Biermann and Feldman 1972], CONTRACTOR++ [Krka et al. 2014], SEKT [Krka et al. 2014], and TEMI [Krka et al. 2014]. Therefore, we omitted comparisons with these approaches.

LE. Weiss et al. [2024] proposed an approach for extracting an FSA from an RNN. They first trained an RNN to recognize positive and negative examples, then applied the L* algorithm to extract an FSA representation from the trained RNN. Their results showed that LE extracted FSAs quickly and accurately with obvious advantages compared to other approaches in many languages.

DICE. Kang and Lo [2021] introduced adversarial FSA mining only from positive examples. Their core idea is to use test generation techniques to synthesize new examples not covered by original examples. Their results showed that they mined FSAs with higher quality than Tautoko [Dallmeier et al. 2010], so we omitted comparisons with Tautoko.

SYM. Roy et al. [2023] constructed constraints characterizing a language smaller FSA and used a constraint solver to find an FSA satisfying the constraints. Their results showed that SYM had significant advantages over other approaches to mining only from positive examples.

DISC. Shvo et al. [2021] modeled the problem of mining FSAs as a MILP. Regularization is performed by limiting the number of states and setting absorbing states to alleviate overfitting. Their results showed that DISC outperformed other approaches to mining from positive and negative examples.

We did not compare with the approaches [Cao and Zhang 2020; Le et al. 2015]. The approach [Le et al. 2015] has been dominated by DSM and DICE and their code is unavailable. The code of the approach [Cao and Zhang 2020] is also not available. Besides, Cao and Zhang [2020] used GTMs to select an optimal FSA during mining FSAs, which is different from ours: we only used GTMs for evaluating, since the GTMs are often difficult to obtain in practice.

6.2 Dataset

6.2.1 Library Classes and GTMs. Following the work [Kang and Lo 2021; Le and Lo 2018], we used 11 library classes as datasets. These library classes were introduced by previous work [Krka et al. 2014; Le et al. 2015]. These library classes have their own corresponding GTMs[2], created by the work [Krka et al. 2014]. Note that these GTMs were extracted manually [Krka et al. 2014] and were continuously refined through a series of works [Le et al. 2015; Le and Lo 2018], and finally were followed by the work [Kang and Lo 2021]. Therefore, their correctness is guaranteed. The details about the datasets are shown in Table 1.

Table 1. The details about datasets, where '$|S|$' means the number of states, '$|\Sigma|$' means the number of APIs, and '$|\delta|$' means the number of state-transitions. NFST is short for 'NumberFormatStringTokenizer'.

dateset	GTM		
	$\|S\|$	$\|\Sigma\|$	$\|\delta\|$
ArrayList	3	16	28
HashMap	3	10	20
HashSet	3	11	18
Hashtable	3	8	15
LinkedList	3	10	20
NFST	5	8	22
Signature	3	4	5
Socket	5	20	51
StackAr	7	9	38
StringTokenizer	3	4	6
ZipOutputStream	4	5	10

6.2.2 Execution Traces. The execution traces are sequences of APIs called in library classes. For every library class, we split the execution traces generated by the work [Le and Lo 2018][3] to build 10 random training and testing sets. We first purely randomly split execution traces into 10 equal parts. Then, each part as a testing set was combined with the remaining 9 parts as a training set to form 10 different pairs of training and testing sets. For every library class, all approaches were trained and tested on 10 different pairs of training and test sets with 5 different random seeds, and average performance was reported. The random seeds were set to $\{1, 2, 3, 4, 5\}$.

6.2.3 Injecting Noise. Following the works [Amar et al. 2018; Bao et al. 2019] on software evolution, we injected noise into training sets. They synthesized new logs by randomly mutating original logs, to simulate generating logs of different software versions. It is reasonable to adapt their method. Because one of the main sources of noise is buggy software [Yang et al. 2006], the different logs generated by bug fixes in software evolution are likely to contain noise. Besides, the rapid software evolution is the motivation to develop automatic FSA mining [Le and Lo 2018].

Specifically, we introduced the noise rate η. For each training set, we first randomly selected η examples. Then, we randomly selected a mutation operation for each selected example and performed mutation operations. Finally, we replaced selected examples with the mutated ones. Mutation operations of a string π are defined as follows.

- Replacement: randomly select a symbol in π, and replace it with other random symbols.
- Insertion: randomly select a symbol in π, and select a random symbol to insert after it.
- Exchange: randomly select two different symbols in π, and exchange their positions.
- Deletion: randomly select a symbol in π, and delete it.

As the noise rate increases, the sample distribution of the original training set may be broken. Also, it leads to the growth of mistakable positive examples. To sum up, the noise injection method can produce noise, and data with a higher noise rate are more challenging for FSA mining approaches. To study the noise-tolerance of approaches, we produced noisy training sets with different $\eta \in \{10\%, 20\%, 30\%, 40\%, 50\%\}$.

[2] https://github.com/kanghj/DICE/tree/master/ground_truth
[3] https://github.com/hvdthong/DSM/tree/master/data

6.3 Setting

6.3.1 Evaluation Metric. We followed the evaluation metrics [Lo and Khoo 2006a], *i.e.*, precision, recall, and F1 score, to measure the quality of mined FSAs. They has been widely adopted by many previous works [Kang and Lo 2021; Le and Lo 2018]. Let $\mathcal{A}^*$ be a GTM and $\mathcal{A}$ a mined FSA. The similarity between $\mathcal{A}^*$ and $\mathcal{A}$ is computed as follows. First, we sampled a set of strings accepted by $\mathcal{A}^*$ (*resp.*, $\mathcal{A}$), denoted by $\Pi_{\mathcal{A}^*}$ (*resp.*, $\Pi_{\mathcal{A}}$). Then, we computed precision (pre.) and recall (rec.) defined as Equation (9). Intuitively, a higher precision indicates that an approach has a better ability to prevent overgeneralizing and a higher recall indicates that an approach has a better ability to alleviate overfitting. We computed F1 score (Equation (9)) as a summary metric.

$$\text{pre.} = \frac{\{\pi \in \Pi_{\mathcal{A}} | \textsc{Accept}(\mathcal{A}^*, \pi) = 1\}}{|\Pi_{\mathcal{A}}|}, \text{rec.} = \frac{\{\pi \in \Pi_{\mathcal{A}^*} | \textsc{Accept}(\mathcal{A}, \pi) = 1\}}{|\Pi_{\mathcal{A}^*}|}, \text{F1} = 2 \times \frac{\text{pre.} \times \text{rec.}}{\text{pre.} + \text{rec.}}. \tag{9}$$

We used the testing sets as samplings of GTMs to ensure that the distribution of the strings in training sets and that of the strings used for evaluation is consistent. For a mined FSA, sampling must thoroughly cover its states and transitions. To this end, we set the maximum number of generated strings to 10, 000 with a maximal size of 50, and the minimum coverage of each transition equaled 20%, which is the same as previous works [Kang and Lo 2021; Le and Lo 2018]. We used running time to measure the efficiency of mining FSAs. For all approaches, the running time includes the time from reading data to returning a mined FSA.

Since we cannot directly sample on `FSAccept`, we cannot evaluate the performance gap between `FSAccept` and the FSA interpreted by its parameters using precision, recall, and F1 score. For every library class, we generated a testing set based on GTMs. The GTM-based testing set includes positive and negative examples, where we used the execution traces generated by the work [Le and Lo 2018] as positive examples. Let L be the maximum size of the positive examples and M be the number of the positive examples. We iteratively generated random strings with a maximum size of L and checked negative examples via the GTM until M negative examples are obtained. On the GTM-based testing set, we calculate $|acc_{\texttt{FSAccept}} - acc_{FSA}|$ to evaluate the performance gap, where $acc_{\texttt{FSAccept}}$ and acc_{FSA} represent the accuracy of `FSAccept` and the FSA interpreted by its parameters, respectively.

6.3.2 Configuration. All approaches first mined an FSA from a training set and then were compared by the similarity between the mined FSA and the GTM. 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 was 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 and using the Adam optimizer. For `DSM` and `DICE`, since they used the same datasets as us, we used the optimal hyper-parameters provided in their work. For `LE`, `SYM`, `DISC`, and our approaches, we employed a standard k-fold cross-validation procedure to select the optimal hyper-parameters, where we set k to 10. K-fold cross-validation is widely used in machine learning for hyper-parameter optimization when a validation set is not available. Note that we did not determine optimal hyper-parameters for each dataset, but rather a unified optimal hyper-parameters for all datasets, which shows that our hyper-parameters are generalizable. Besides, each approach introduces different hyper-parameters based on its own characteristics, making it difficult to conduct a completely fair comparison with identical parameters. However, from the perspective of using optimal parameters, our comparison is fair. The main hyper-parameters of `NADA` are set as follows: $\alpha = 0.01$; $\beta = 2$; $\gamma = 0.8$; $N_{\text{s}}^{\text{m}} = 5$; the learning rate is 0.01; the the maximum epoch is 325 and the batch size is 128.

Table 2. Experimental results (%) across different approaches, where 'average' means the mean of the metric over all datasets, 'time' means the running time (s), and **boldface** numbers refer to the better results.

dateset	DSM		LE		DICE		SYM		DISC		NADA	
	pre.	rec.	pre.	rec.	pre.	rec.	pre.	rec.	pre.	rec.	pre.	rec.
ArrayList	27.40	31.70	8.00	83.11	**84.00**	25.80	26.00	**100.00**	10.40	90.77	69.48	89.38
HashMap	**100.00**	87.10	15.60	99.93	**100.00**	85.50	**100.00**	**100.00**	34.20	98.29	**100.00**	91.66
HashSet	67.40	75.60	9.80	99.88	64.40	91.10	23.00	**99.90**	3.00	97.47	**100.00**	89.08
Hashtable	96.50	73.80	18.80	99.72	72.50	92.40	4.00	**100.00**	12.60	99.05	**100.00**	89.50
LinkedList	**100.00**	10.00	18.60	89.76	**100.00**	37.70	**100.00**	**100.00**	12.40	99.52	**100.00**	91.68
NFST	55.90	39.90	14.60	91.32	92.50	66.20	11.00	**100.00**	4.20	98.22	**100.00**	95.14
Signature	**100.00**	99.10	10.00	**100.00**	**100.00**	99.10	**100.00**	**100.00**	35.60	**100.00**	**100.00**	**100.00**
Socket	41.90	66.10	7.20	92.72	72.50	29.70	44.00	**100.00**	4.00	93.21	**91.96**	92.72
StackAr	24.60	98.40	15.60	97.63	92.50	87.40	14.00	**100.00**	4.20	93.48	**100.00**	95.90
StringTokenizer	92.50	98.80	29.20	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**	6.80	91.00	**100.00**	**100.00**
ZipOutputStream	**100.00**	99.40	20.40	**100.00**	**100.00**	91.60	**100.00**	**100.00**	1.00	**100.00**	97.66	**100.00**
average	73.29	70.90	15.25	95.83	88.95	73.32	56.55	**99.99**	11.67	96.45	**96.28**	94.10

dateset	DSM		LE		DICE		SYM		DISC		NADA	
	F1	time	F1	time	F1	time	F1	time	F1	time	F1	time
ArrayList	28.78	31.22	14.60	835.88	39.34	3428.10	41.27	**11.95**	18.66	966.88	**77.60**	855.93
HashMap	93.06	53.67	26.99	409.71	92.17	2138.68	**100.00**	**13.68**	50.74	720.86	95.00	769.38
HashSet	68.46	198.34	17.85	1852.44	75.19	4712.52	37.39	**10.82**	5.82	11348.40	**93.40**	2049.30
Hashtable	83.45	74.22	31.64	522.69	81.19	8505.58	7.69	**1.83**	22.36	1474.56	**94.00**	668.95
LinkedList	17.91	10.95	30.81	300.74	54.70	940.05	**100.00**	**7.21**	22.05	59.90	95.40	500.01
NFST	42.88	82.95	25.18	531.36	77.03	156002.13	19.82	**4.53**	8.06	4676.28	**97.40**	804.78
Signature	99.55	163.60	18.18	263.32	99.55	1480.75	**100.00**	**3.10**	52.51	4476.97	**100.00**	480.67
Socket	49.23	128.89	13.36	2985.46	41.52	27283.12	61.11	**28.00**	7.67	12380.29	**92.00**	3258.35
StackAr	38.68	241.01	26.90	874.22	89.73	1678.27	24.56	**15.49**	8.04	9551.67	**97.00**	930.09
StringTokenizer	94.98	40.90	45.20	533.61	**100.00**	1208.82	**100.00**	**5.33**	12.65	10106.21	**100.00**	572.34
ZipOutputStream	99.70	103.01	33.89	137.74	95.61	15426.04	**100.00**	**2.49**	1.98	2610.88	98.60	360.85
average	65.15	102.61	25.87	840.65	76.91	20254.91	62.90	**9.49**	19.14	5306.63	**94.58**	1022.79

6.4 Result Analysis

6.4.1 Compared with SOTA Approaches. As shown in Table 2, although NADA spends more running time than DSM, NADA outperforms DSM in terms of both precision and recall on all datasets except StackAr and ZipOutputStream. The average precision, recall, and F1 score of all datasets significant improvements are 22.99%, 23.20%, and 29.43% respectively. Compared with LE, NADA improves on the average precision and F1 score by over 81.03% and 68.71%. It shows the vulnerability of LE when dealing with noise, and proves the performance of NADA in alleviating overgeneralizing compared to LE. LE is faster than NADA because they converge to a local optimum quickly when training an RNN, which may also be the reason for its overgeneralizing. Compared with SYM, NADA shows an improvement of over 39.73% and 31.68% respectively on the average precision and F1 score. In some datasets, *e.g.*, HashMap and LinkedList, SYM displays outstanding performance, while it does not achieve good performance in some datasets such as Hashtable, NFST, and StackAr. In terms of running time, SYM is extremely faster than NADA, benefiting from the excellent performance of modern SAT solvers. Compared with DISC, NADA achieves remarkable improvement, with an increase of over 84.61% and 75.44% respectively in terms of precision and F1 score. Although DISC can also utilize both positive and negative examples, it shows vulnerability in resisting noisy data. Regarding running time, DISC spends much more time (5.19X) than NADA due to the performance restriction of MILP solvers. DICE mines FSAs with the highest quality among all baselines, but its time cost is the largest. Compared with DICE, NADA still on average improves the precision, recall, and F1 score by 7.33%, 20.78%, and 17.67%, respectively. In terms of the running time, DICE spends much more time (19.8X) than NADA.

Table 3. Ablation results (%) about the faithful FSA encoding and the potential negative examples, where 'w/o FE & PNE' indicates it without faithful FSA encoding and potential negative examples, 'w/o FE' indicates it without faithful FSA encoding, 'w/o PNE' indicates it without potential negative examples, '-RSPNE' indicates it equipped with Algorithm 8, and 'TN/PN' indicates the percentage of true negative examples in potential negative examples (%).

dateset	NADA w/o FE and PNE			NADA w/o FE			NADA w/o PNE			NADA-RSPNE				NADA			
	pre.	rec.	F1	pre.	rec.	F1	pre.	rec.	F1	pre.	rec.	F1	TN/PN	pre.	rec.	F1	TN/PN
ArrayList	**100.00**	0.00	0.00	78.10	67.30	72.00	19.90	**100.00**	33.00	27.60	42.60	33.00	14.20	69.48	89.38	**77.60**	**48.50**
HashMap	0.00	0.00	0.00	**100.00**	91.90	95.00	**100.00**	**100.00**	**100.00**	**100.00**	76.50	86.00	9.50	**100.00**	91.66	95.00	**84.20**
HashSet	0.00	0.00	0.00	**100.00**	85.10	91.00	18.60	**100.00**	31.00	20.40	69.30	31.00	11.00	**100.00**	89.08	**93.40**	**72.60**
Hashtable	0.00	0.00	0.00	**100.00**	89.60	**94.00**	17.10	**100.00**	29.00	21.40	50.50	30.00	11.20	**100.00**	89.50	**94.00**	**88.10**
LinkedList	0.00	0.00	0.00	**100.00**	79.80	88.00	**100.00**	**100.00**	**100.00**	**100.00**	60.70	75.00	8.70	**100.00**	91.68	95.40	**43.80**
NFST	0.00	0.00	0.00	97.90	**100.00**	**98.00**	4.80	**100.00**	9.00	6.50	18.00	9.00	12.60	**100.00**	95.14	97.40	**94.00**
Signature	**100.00**	0.00	0.00	**100.00**	**100.00**	**100.00**	34.10	**100.00**	50.00	**100.00**	96.90	98.00	24.00	**100.00**	**100.00**	**100.00**	**100.00**
Socket	**100.00**	0.00	0.00	42.80	76.30	54.00	9.20	**100.00**	16.00	12.20	59.20	20.00	17.30	91.96	92.72	**92.00**	**91.70**
StackAr	**100.00**	0.00	0.00	83.30	96.80	89.00	5.80	**100.00**	11.00	40.10	11.00	17.00	15.10	**100.00**	95.90	**97.00**	**100.00**
StringTokenizer	**100.00**	0.00	0.00	**100.00**	**100.00**	**100.00**	2.20	**100.00**	4.00	9.40	36.90	15.00	22.00	**100.00**	**100.00**	**100.00**	**100.00**
ZipOutputStream	0.00	0.00	0.00	**100.00**	**100.00**	**100.00**	2.10	**100.00**	4.00	**100.00**	99.30	99.00	32.40	97.66	**100.00**	98.60	**93.60**
average	45.45	0.00	0.00	91.10	89.71	89.18	28.53	**100.00**	35.18	48.87	56.45	46.64	16.18	**96.28**	94.10	**94.58**	**83.32**

To sum up, NADA mines higher quality FSAs than all SOTA approaches, and it is much faster than DICE, an approach that mines sub-high-quality FSAs. These conclusions answer RQ 1. Besides, from the same set of positive and negative examples with noise, NADA is able to mine higher quality FSAs than LE and DISC, indicating that NADA has a stronger noise tolerance.

6.4.2 Ablation Study. From the results in Table 3, the performance of NADA is obviously higher, especially compared with NADA w/o FE & PNE, which shows that both the faithful FSA encoding and the potential negative examples have positive impacts. NADA w/o FE & PNE achieves the lowest performance. Table 4 shows that in most cases there is a huge performance gap between a trained FSAccept w/o FE & PNE and its interpreted FSA. We checked the trained parameters of FSAccept w/o FE & PNE and found that a large number of values of transition matrices and accepting vectors are slightly less than 0.5. In this case, although FSAccept can accept examples, the interpreted FSA lacks accepting states and transitions from the initial state to other states, making the interpreted FSA reject all examples. Such a case is caused by the accumulated errors in computing DifAccept. Besides, the results that NADA w/o FE & PNE reaches 100% precision on some datasets do not imply that the interpreted FSA is high-quality. The reason why NADA w/o FE & PNE can reach 100% precision is that the interpreted FSA has only one or a few transitions to an accepting state, so that almost all strings sampled from the interpreted FSA are accepted by the GTM.

Algorithm 8: Randomly mutate positive examples.

Input : A positive example π.
Output : A potential negative example π'.

$seed \leftarrow$ randomly get a random integer in $\{1, 2, 3, 4, 5, 6\}$
if $seed = 1$ **then**
  $\pi' \leftarrow$ randomly select $p, q \in \pi$, and remove all q following p
else if $seed = 2$ **then**
  $\pi' \leftarrow$ randomly select $p, q \in \pi$, and insert q at a random position following p
else if $seed = 3$ **then**
  $\pi' \leftarrow$ randomly select a $p \in \pi$, randomly select the $q \in \pi$ preceding the first p in π, and exchange q and p
else if $seed = 4$ **then**
  $\pi' \leftarrow$ randomly select a $p \in \pi$, and remove the q immediately following p
else if $seed = 5$ **then**
  $\pi' \leftarrow$ randomly select $p, q \in \pi$, and insert a q immediately following p
else if $seed = 6$ **then**
  $\pi' \leftarrow$ randomly select a $p \in \pi$, select the $q \in \pi$ immediately preceding the first p in π, and exchange q and p
return π'

From the comparison results between NADA w/o FE & PNE and NADA w/o PNE as well as that between NADA and NADA w/o FE in terms of recall, it shows that the faithful FSA encoding helps alleviate overfitting. The results in Table. 4 indicate that the faithful FSA encoding benefits to a high performance of FSAccept, because it softly constrains trained parameters corresponding one to one with FSA, thereby alleviating errors caused by the interpretation. Similarly, from the comparison results between NADA and NADA w/o PNE in terms of precision, we conclude that sampling potential negative examples helps prevent overgeneralizing. The column 'TN / PN' in Table 3 demonstrates that SynthesizePNE is reasonable since on average over 80% of the potential negative examples are true negative examples. We also noticed two datasets with lower TN/PN (ArrayList and LinkedList), where NADA is also better or comparable to NADA w/o PNE in precision. It indicates that our approach exhibits a certain degree of robustness against noise. The comparison results between NADA w/o FE & PNE and NADA w/o FE also basically support the above conclusion.

Table 4. The performance gap (%) between a trained FSAccept and the FSA interpreted from its parameters.

dateset	w/o FE and PNE	w/o FE	w/o PNE	NADA
ArrayList	24.60	14.80	**0.90**	6.30
HashMap	50.00	4.30	**0.00**	4.90
HashSet	24.10	**0.90**	10.60	4.50
Hashtable	27.90	5.70	**0.10**	2.40
LinkedList	50.00	4.20	**0.00**	3.00
NFST	12.90	1.30	21.40	**0.10**
Signature	50.00	**0.00**	**0.00**	**0.00**
Socket	7.50	5.50	6.40	**0.60**
StackAr	16.90	3.70	2.00	**1.30**
StringTokenizer	18.80	**0.00**	10.10	**0.00**
ZipOutputStream	18.70	15.70	2.10	**0.00**
average	27.40	5.10	4.87	**2.10**

To evaluate SynthesizePNE, we considered a random synthesis method without the guidance by LTL formulae. Specifically, we replaced Algorithm 4 in SynthesizePNE with Algorithm 8, denoted as RSPNE. It is similar to SynthesizePNE but does not use LTL formulae to guide the synthesis of negative examples. Since a random synthesis method does not need to generate LTL formulae, it will be more efficient. However, Table 3 shows that NADA significantly outperforms NADA-RSPNE in terms of both precision and recall, demonstrating the effectiveness of SynthesizePNE. The TN/PN value of NADA is higher than that of NADA-RSPNE, indicating that this effectiveness mainly comes from the synthesis of more true negative examples.

In summary, considering faithful FSA encoding in training FSAccept to fit the training set is able to alleviate overfitting and benefits to mining high-quality FSA; furthermore, adding potential negative examples can prevent overgeneralizing. These conclusions answer RQ 2.

6.4.3 Impact of Noise. We omitted the results of LE and DISC because they are poor on all datasets with injected noise (F1 score below 20%). We report the percentage of false positive and false negative examples resulting from noise among all training examples in Fig. 2. Overall, as the noise rate increases, more false positive examples will be introduced. Fig. 3 shows that NADA outperforms DSM, DICE, and SYM notably on datasets with different η. As the noise rate increases, the advantage of NADA becomes more obvious as the precision of NADA remains high. NADA is robust against different noise rates.

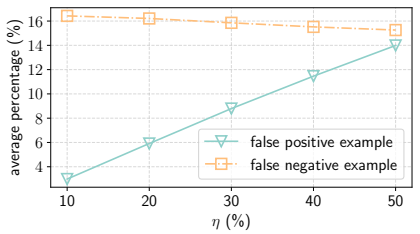


Fig. 2. Average percentage of false positive and false negative examples.

To sum up, NADA is more noise-tolerant than the SOTA approaches, which answers RQ 3.

6.4.4 Robustness to Hyper-parameters. The main hyper-parameters in NADA are β, γ, and $N_s^{\mathbf{m}}$. We conducted experiments to show the impact of different values of these hyper-parameters, as illustrated in Fig. 4. The setting of β has no obvious impact on the results. We observe that, when

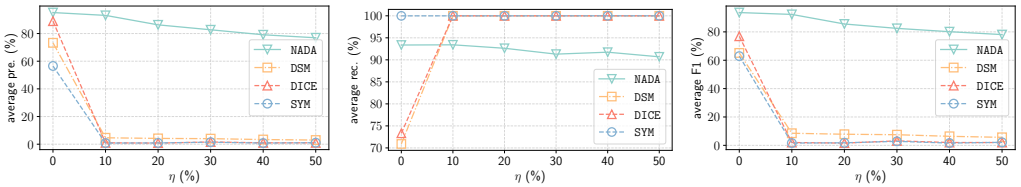


Fig. 3. Experimental results on noise data. The results are the average results of all datasets.

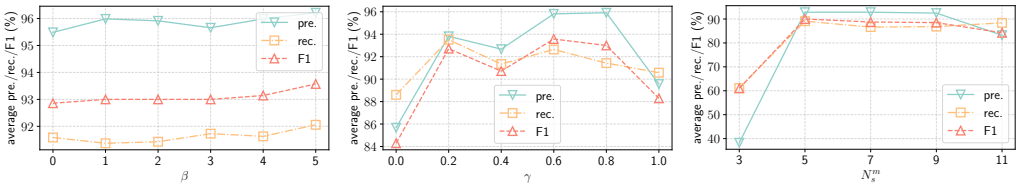


Fig. 4. Comparisons for different settings of β, γ, and N_s^m. The results are the average results of all datasets.

$\beta = 0$ or $= 1$, it is possible to mine FSAs without accepting states, resulting in lower performance. Therefore, we set $\beta = 2$ to improve the robustness. As γ increases, the performance improves and eventually drops after $\gamma = 0.8$. The performance gap between $\gamma = 0.8$ and $\gamma = 0.6$ is not significant. Except for the case where $N_s^m = 3$, N_s^m has little impact on performance. It shows that our approach has a certain degree of robustness to N_s^m.

To sum up, NADA is robust to hyper-parameters within a certain range, which answers RQ 4.

6.4.5 Case Study. We examine why all compared approaches perform poorly on ArrayList. We refer to a pair of consecutive symbols as an adjacent transition. An adjacent transition determines an incoming transition and an outgoing transition of a state in an FSA. Fig. 5 shows the coverage rate of all positive examples for each adjacent transition in the GTM of ArrayList. The results show that the coverage rate is unevenly distributed and a large number of adjacent transitions are not present in the data. This distribution misleads the mining approaches to ignore those transitions with a low coverage rate.

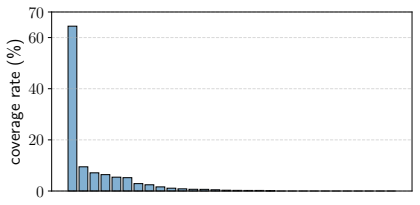


Fig. 5. Case study on ArrayList, where the Y axis is the coverage rate, and the X axis denotes adjacent transitions.

Although SYM is the fastest among all compared approaches, it performs poorly on some datasets, particularly for Hashtable. By looking deep into the GTM of Hashtable, we discover that the FSA mined by SYM has several redundant transitions. This indicates that, due to the lack of negative examples, SYM results in overgeneralization.

7 Threat to Validity

The first threat comes from the scalability of FSA size, which depends on the hyper-parameter N_s^m. In practice, the number of states of a GTM is unknown, which makes it challenging to reasonably set N_s^m. If N_s^m is set too large, it will lead to many trainable parameters, making the training much harder; if N_s^m is set too small, it will make neural networks inexpressive to model a GTM. Fortunately, experimental results show that when N_s^m is larger than the number of states of a

GTM within a certain range, the performance of our approach does not fluctuate too much (Fig. 4). Therefore, in practice, we recommend using a large $N_s^{\mathbf{m}}$ to ensure higher performance.

The second threat is the expressiveness of template-constrained LTL formulae. Some works [Kang and Lo 2021; Sun et al. 2019] have found that the expressiveness of existing template-constrained LTL formulae is insufficient. The reason for not learning LTL fragments with higher expressivity is due to efficiency. As advanced methods for learning LTL formulas appear, we will consider learning arbitrary-form formulae via efficient approaches [Luo et al. 2022; Valizadeh et al. 2024; Wan et al. 2024] in future work.

The uneven sampling distribution of strings is the third threat for FSA mining [Kang and Lo 2021]. For StringTokenizer, we observe that the GTM can accept $STNHMTF$, but only 0.064% of the strings in the training set adopt this pattern. Such an uneven sampling distribution will seriously impair the performance of mining approaches. Kang and Lo [2021] introduced the idea of adversarial mining to tackle this issue. Their approach is orthogonal to ours, and thus can be combined with ours to further improve performance.

The last threat is the accumulation of errors in neural acceptance. Since we perform continuous relaxation on DifAccept during the training process, the errors will be amplified as the size of the string increases. Although this threat is not obvious yet, in order to extend our approach to strings with large sizes, we will exploit the optimizations provided by [Wang et al. 2021] or train neural networks for early classification [Shvo et al. 2021] in future work.

8 Conclusion and Future Work

Our work points towards a fresh mining paradigm to mine high-quality FSAs via modeling FSAs in the discrete domain as learning parameters in the continuous domain. Its insight is to bridge FSA acceptance to neural acceptance by an FSA encoding method, which is the basis for designing a neural network to simulate FSA acceptance. It efficiently searches for FSAs via gradient descent-based training of neural networks and is easily combined with potential negative examples to alleviate overgeneralization. Experimental results have demonstrated that our approach significantly improves the quality of mined FSAs and is robust to noise.

Future work will evaluate the effectiveness of our approach in industrial scenarios and extend our approach to formalize properties of data in verified AI software.

9 Data Availability

Our code and datasets are publicly available at: https://github.com/sysulic/NADA.

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, 2025A1515011639), as well as Shenzhen Science and Technology Program (KJZD2023092311405902).

References

Hen Amar, Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. 2018. Using finite-state models for log differencing. In *FSE*. 49–59. doi:10.1145/3236024.3236069

Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *POPL*. 4–16. doi:10.1145/503272.503275

Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106. doi:10.1016/0890-5401(87)90052-6

Dana Angluin, Sarah Eisenstat, and Dana Fisman. 2015. Learning Regular Languages via Alternating Automata. In *IJCAI*. 3308–3314.

Florent Avellaneda and Alexandre Petrenko. 2018. Inferring DFA without Negative Examples. In *ICGI*, Vol. 93. 17–29.

Stéphane Ayache, Rémi Eyraud, and Noé Goudian. 2018. Explaining Black Boxes on Sequential Data using Weighted Automata. In *ICGI*, Vol. 93. 81–103.

Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. 2019. Statistical Log Differencing. In *ASE*. 851–862. doi:10.1109/ASE.2019.00084

Andreas Bauer, Martin Leucker, and Christian Schallhart. 2007. The Good, the Bad, and the Ugly, But How Ugly Is Ugly?. In *RV*, Vol. 4839. 126–138. doi:10.1007/978-3-540-77395-5_11

Leonard E Baum and Ted Petrie. 1966. Statistical inference for probabilistic functions of finite state Markov chains. *The annals of mathematical statistics* 37, 6 (1966), 1554–1563.

Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2015. Using Declarative Specification to Improve the Understanding, Extensibility, and Comparison of Model-Inference Algorithms. *IEEE Trans. Software Eng.* 41, 4 (2015), 408–428. doi:10.1109/TSE.2014.2369047

Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *FSE*. 267–277. doi:10.1145/2025113.2025151

Alan W. Biermann and Jerome A. Feldman. 1972. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Computers* 21, 6 (1972), 592–597. doi:10.1109/TC.1972.5009015

Jonathon Cai, Richard Shin, and Dawn Song. 2017. Making Neural Programming Architectures Generalize via Recursion. In *ICLR*. OpenReview.net.

Alberto Camacho and Sheila A. McIlraith. 2019. Learning Interpretable Models Expressed in Linear Temporal Logic. In *ICAPS*. 621–630.

Zhi Cao and Nan Zhang. 2020. Deep Specification Mining with Attention. In *COCOON*, Vol. 12273. 186–197. doi:10.1007/978-3-030-58150-3_15

Adelmo Luis Cechin, Denise Regina Pechmann Simon, and Klaus Stertz. 2003. State Automata Extraction from Recurrent Neural Nets using k-Means and Fuzzy Clustering. In *SCCC*. 73–78. doi:10.1109/SCCC.2003.1245447

Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. 2010. Generating test cases for specification mining. In *ISSTA*. 85–96. doi:10.1145/1831708.1831719

Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. 2006. Mining object behavior with ADABU. In *Workshop on DSA*. 17–24. doi:10.1145/1138912.1138918

Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. 2012. Automated Abstractions for Contract Validation. *IEEE Trans. Software Eng.* 38, 1 (2012), 141–162. doi:10.1109/TSE.2010.98

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in Property Specifications for Finite-State Verification. In *ICSE*. 411–420. doi:10.1145/302405.302672

Jean-Raphaël Gaglione, Daniel Neider, Rajarshi Roy, Ufuk Topcu, and Zhe Xu. 2021. Learning Linear Temporal Properties from Noisy Data: A MaxSAT-Based Approach. In *ATVA*, Vol. 12971. 74–90. doi:10.1007/978-3-030-88885-5_6

Georgios Giantamidis, Stavros Tripakis, and Stylianos Basagiannis. 2021. Learning Moore machines from input-output traces. *Int. J. Softw. Tools Technol. Transf.* 23, 1 (2021), 1–29. doi:10.1007/S10009-019-00544-0

E. Mark Gold. 1967. Language Identification in the Limit. *Inf. Control.* 10, 5 (1967), 447–474. doi:10.1016/S0019-9958(67)91165-5

E. Mark Gold. 1978. Complexity of Automaton Identification from Given Data. *Inf. Control.* 37, 3 (1978), 302–320. doi:10.1016/S0019-9958(78)90562-4

Petr Grachev, Igor Lobanov, Ivan Smetannikov, and Andrey Filchenkov. 2017. Neural network for synthesizing deterministic finite automata. *Procedia computer science* 119 (2017), 73–82.

Marijn Heule and Sicco Verwer. 2010. Exact DFA Identification Using SAT Solvers. In *ICGI*, Vol. 6339. 66–79. doi:10.1007/978-3-642-15488-1_7

Henrik Jacobsson. 2005. Rule Extraction from Recurrent Neural Networks: A Taxonomy and Review. *Neural Comput.* 17, 6 (2005), 1223–1263. doi:10.1162/0899766053630350

Hong Jin Kang and David Lo. 2021. Adversarial Specification Mining. *ACM Trans. Softw. Eng. Methodol.* 30, 2 (2021), 16:1–16:40. doi:10.1145/3424307

John F. Kolen. 1993. Fool's Gold: Extracting Finite State Machines from Recurrent Network Dynamics. In *NeurIPS*. Morgan Kaufmann, 501–508.

Anurag Koul, Alan Fern, and Sam Greydanus. 2019. Learning Finite State Representations of Recurrent Policy Networks. In *ICLR*.

Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic mining of specifications from invocation traces and method invariants. In *FSE*. 178–189. doi:10.1145/2635868.2635890

Kevin J. Lang. 1992. Random DFA's Can Be Approximately Learned from Sparse Uniform Examples. In *COLT*. 45–52. doi:10.1145/130385.130390

Kevin J Lang. 1999. Faster algorithms for finding minimal consistent DFAs. *NEC Research Institute, Tech. Rep* (1999).

Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. 1998. Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. In *ICGI*, Vol. 1433. 1–12. doi:10.1007/BFB0054059

Tien-Duy B. Le, Xuan-Bach Dinh Le, David Lo, and Ivan Beschastnikh. 2015. Synergizing Specification Miners through Model Fissions and Fusions (T). In *ASE*. 115–125. doi:10.1109/ASE.2015.83

Tien-Duy B. Le and David Lo. 2018. Deep specification mining. In *ISSTA*. 106–117. doi:10.1145/3213846.3213876

Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL Specification Mining (T). In *ASE*. 81–92. doi:10.1109/ASE.2015.71

David Lo and Siau-Cheng Khoo. 2006a. QUARK: Empirical Assessment of Automaton-based Specification Miners. In *WCRE*. 51–60. doi:10.1109/WCRE.2006.47

David Lo and Siau-Cheng Khoo. 2006b. SMArTIC: towards building an accurate, robust and scalable specification miner. In *FSE*. 265–275. doi:10.1145/1181775.1181808

David Lo, Leonardo Mariani, and Mauro Pezzè. 2009. Automatic steering of behavioral model inference. In *FSE*. 345–354. doi:10.1145/1595696.1595761

Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. In *ICSE*. 501–510. doi:10.1145/1368088.1368157

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. doi:10.1609/AAAI.V36I9.21221

Mingjun Ma, Dehui Du, Yuanhao Liu, Yanyun Wang, and Yiyang Li. 2022. Efficient Adversarial Sequence Generation for RNN with Symbolic Weighted Finite Automata. In *AAAI*, Vol. 3087.

Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. 2011. Dynamic Analysis for Diagnosing Integration Faults. *IEEE Trans. Software Eng.* 37, 4 (2011), 486–508. doi:10.1109/TSE.2010.93

Weikai Miao and Shaoying Liu. 2012. A Formal Specification-Based Integration Testing Approach. In *SOFL*, Vol. 7787. 26–43. doi:10.1007/978-3-642-39277-1_3

Joshua J. Michalenko, Ameesh Shah, Abhinav Verma, Richard G. Baraniuk, Swarat Chaudhuri, and Ankit B. Patel. 2019. Representing Formal Languages: A Comparison Between Finite Automata and Recurrent Neural Networks. In *ICLR*.

Alexander Mordvintsev. 2022. Differentiable Finite State Machines. https://google-research.github.io/self-organising-systems/2022/diff-fsm/

Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through hoops: why do Java developers struggle with cryptography APIs?. In *ICSE*. 935–946. doi:10.1145/2884781.2884790

Daniel Neider and Ivan Gavran. 2018. Learning Linear Temporal Properties. In *FMCAD*. 1–10. doi:10.23919/FMCAD.2018.8603016

Takamasa Okudono, Masaki Waga, Taro Sekiyama, and Ichiro Hasuo. 2020. Weighted Automata Extraction from Recurrent Neural Networks via Regression on State Spaces. In *AAAI*. 5306–5314. doi:10.1609/AAAI.V34I04.5977

Arlindo L. Oliveira and João P. Marques Silva. 2001. Efficient Algorithms for the Inference of Minimum Size DFAs. *Mach. Learn.* 44, 1/2 (2001), 93–119. doi:10.1023/A:1010828029885

Christian W. Omlin and C. Lee Giles. 1996a. Extraction of rules from discrete-time recurrent neural networks. *Neural Networks* 9, 1 (1996), 41–52. doi:10.1016/0893-6080(95)00086-0

Christian W. Omlin and C. Lee Giles. 1996b. Rule Revision With Recurrent Neural Networks. *IEEE Trans. Knowl. Data Eng.* 8, 1 (1996), 183–188. doi:10.1109/69.485647

Christian W. Omlin, Karvel K. Thornber, and C. Lee Giles. 1996. Representation of fuzzy finite state automata in continuous recurrent, neural networks. In *ICNN*. IEEE, 1023–1027. doi:10.1109/ICNN.1996.549038

José Oncina and Pedro Garcia. 1992. Identifying regular languages in polynomial time. In *Advances in structural and syntactic pattern recognition*. 99–108.

José Oncina, Pedro Garcia, et al. 1992. Inferring regular languages in polynomial update time. *Pattern recognition and image analysis* 1, 49-61 (1992), 10–1142.

Leonard Pitt and Manfred K. Warmuth. 1993. The Minimum Consistent DFA Problem Cannot be Approximated within any Polynomial. *J. ACM* 40, 1 (1993), 95–142. doi:10.1145/138027.138042

Amir Pnueli. 1977. The Temporal Logic of Programs. In *FOCS*. 46–57. doi:10.1109/SFCS.1977.32

Rajarshi Roy, Jean-Raphaël Gaglione, Nasim Baharisangari, Daniel Neider, Zhe Xu, and Ufuk Topcu. 2023. Learning Interpretable Temporal Properties from Positive Examples Only. In *AAAI*. 6507–6515. doi:10.1609/AAAI.V37I5.25800

Ingo Schellhammer, Joachim Diederich, Michael Towsey, and Claudia Brugman. 1998. Knowledge Extraction and Recurrent Neural Networks: An Analysis of an Elman Network trained on a Natural Language Learning Task. In *NeMLaP/CoNLL*. 73–78.

Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. 2022. Toward verified artificial intelligence. *Commun. ACM* 65, 7 (2022), 46–55. doi:10.1145/3503914

Maayan Shvo, Andrew C. Li, Rodrigo Toro Icarte, and Sheila A. McIlraith. 2021. Interpretable Sequence Classification via Discrete Optimization. In *AAAI*. 9647–9656. doi:10.1609/AAAI.V35I11.17161

Rick Smetsers, Paul Fiterau-Brostean, and Frits W. Vaandrager. 2018. Model Learning as a Satisfiability Modulo Theories Problem. In *LATA*, Vol. 10792. 182–194. doi:10.1007/978-3-319-77313-1_14

John Stogin, Ankur Arjun Mali, and C. Lee Giles. 2024. A provably stable neural network Turing Machine with finite precision and time. *Inf. Sci.* 658 (2024), 120034. doi:10.1016/J.INS.2023.120034

Peng Sun, Chris Brown, Ivan Beschastnikh, and Kathryn T. Stolee. 2019. Mining Specifications from Documentation using a Crowd. In *SANER*. 275–286. doi:10.1109/SANER.2019.8668025

Kaito Suzuki, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara. 2021. Query Learning Algorithm for Symbolic Weighted Finite Automata. In *ICGI*, Vol. 153. 202–216.

Anej Svete, Robin Chan, and Ryan Cotterell. 2024. On Efficiently Representing Regular Languages as RNNs. In *Findings of ACL*. 4118–4135. doi:10.18653/V1/2024.FINDINGS-ACL.244

Boris Avraamovich Trakhtenbrot and Ya M Barzdin. 1973. *Finite automata: Behavior and synthesis*. Elsevier.

Vladimir Ulyantsev, Ilya Zakirzyanov, and Anatoly Shalyto. 2015. BFS-Based Symmetry Breaking Predicates for DFA Identification. In *LATA*, Vol. 8977. 611–622. doi:10.1007/978-3-319-15579-1_48

Mojtaba Valizadeh, Nathanaël Fijalkow, and Martin Berger. 2024. LTL Learning on GPUs. In *CAV*, Vol. 14683. 209–231. doi:10.1007/978-3-031-65633-0_10

Neil Walkinshaw and Kirill Bogdanov. 2008. Inferring Finite-State Models with Temporal Constraints. In *ASE*. 248–257. doi:10.1109/ASE.2008.35

Hai Wan, Pingjia Liang, Jianfeng Du, Weilin Luo, Rongzhen Ye, and Bo Peng. 2024. End-to-End Learning of LTLf Formulae by Faithful LTLf Encoding. In *AAAI*. 9071–9079. doi:10.1609/AAAI.V38I8.28757

Qinglong Wang, Kaixuan Zhang, Alexander G. Ororbia II, Xinyu Xing, Xue Liu, and C. Lee Giles. 2018. An Empirical Evaluation of Rule Extraction from Recurrent Neural Networks. *Neural Comput.* 30, 9 (2018). doi:10.1162/NECO_A_01111

Xuan Wang, Zheyu Yan, Chang Meng, Yiyu Shi, and Weikang Qian. 2023. DASALS: Differentiable Architecture Search-Driven Approximate Logic Synthesis. In *ICCAD*. 1–9. doi:10.1109/ICCAD57390.2023.10323820

Zhuo Wang, Wei Zhang, Ning Liu, and Jianyong Wang. 2021. Scalable Rule-Based Representation Learning for Interpretable Classification. In *NeurIPS*. 30479–30491.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples. In *ICML*, Vol. 80. 5244–5253.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2019. Learning Deterministic Weighted Automata with Queries and Counterexamples. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 8558–8569.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2024. Extracting automata from recurrent neural networks using queries and counterexamples (extended version). *Mach. Learn.* 113, 5 (2024), 2877–2919. doi:10.1007/S10994-022-06163-2

Yinxing Xue, Junjie Wang, Yang Liu, Hao Xiao, Jun Sun, and Mahinthan Chandramohan. 2015. Detection and classification of malicious JavaScript via attack behavior modelling. In *ISSTA*. 48–59. doi:10.1145/2771783.2771814

Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*. 282–291. doi:10.1145/1134285.1134325

Shiwen Yu, Zengyu Liu, Ting Wang, and Ji Wang. 2024. Neural Solving Uninterpreted Predicates with Abstract Gradient Descent. *ACM Trans. Softw. Eng. Methodol.* 33, 8 (2024), 215:1–215:47. doi:10.1145/3675394

Ilya Zakirzyanov, António Morgado, Alexey Ignatiev, Vladimir Ulyantsev, and João Marques-Silva. 2019. Efficient Symmetry Breaking for SAT-Based Minimum DFA Inference. In *LATA*, Vol. 11417. 159–173. doi:10.1007/978-3-030-13435-8_12

Xiyue Zhang, Xiaoning Du, Xiaofei Xie, Lei Ma, Yang Liu, and Meng Sun. 2021. Decision-Guided Weighted Automata Extraction from Recurrent Neural Networks. In *AAAI*. 11699–11707. doi:10.1609/AAAI.V35I13.17391

Received 2025-02-28; accepted 2025-03-31