Abstract. We present and evaluate techniques for specification, verification, and synthesis of solutions to automata theory problems. We show how these techniques can be useful for aspects of an intelligent tutoring system such as checking of solutions, problem- and solution- generation. We focus on two classical problems: constructing DFAs of regular languages, and proving non-regularity of non-regular languages. We introduce the notion of problem description languages to specify automata problems. We present techniques to verify solutions given as automata or non-regularity proofs. The techniques are based on an improved algorithm for model checking the $\text{EF}^2$ logic. We combine these techniques with the inductive learning approach to solve problems automatically. Furthermore, we show how to leverage the solution synthesis methods to generate variants of a seed problem. We present an experimental evaluation of our techniques on the 100+ problems that we collected from books and courses on automata theory. We show that our techniques can check solutions, and generate new solutions and problems, in real time.

1 Introduction

Intelligent Tutoring Systems (ITS) hold the promise of reinventing education by providing immediate and customized feedback to students without intervention from a human teacher. Recent and widespread access to computational devices, cross-disciplinary technological innovations, and the diverse education needs have created the perfect setting for investment into ITS for various subject domains. The key domain-specific technical components of an ITS are: (a) automated grading, (b) solution or hint generation, and (c) problem generation. Development of these components is a new application area for programming language theory, where traditional programming language theory concepts about specification, verification, and synthesis of systems can play a foundational role.

Strong recent enthusiasm for leveraging online platforms for education (e.g., Khan Academy, Udacity, edX, and Coursera) has further reiterated need for development of ITSs for two additional reasons:4 First, the class size increases

from tens to tens of thousands of students. Hence, automated grading technology is required to grade non-multiple choice questions. Second, the classes are no longer tightly synchronized. Hence, problem generation technology is required to overcome the challenge of published solutions to small problem sets. In addition, solution generation technology is necessary to generate solutions to new problems.

Our focus in this paper is on finite automata theory. The importance of finite state automata in computer science education hardly needs justification. It is a natural concept, rich in structure and potential applications. It arises in diverse settings such as control theory, text editors and lexical analyzers, and models of software interfaces, among others. In particular, we deal with the two most common problems occurring in teaching finite automata theory.

- **Automata construction problems.** These require constructing a deterministic finite automaton (DFA) for a regular language.
- **Non-regularity proof problems.** These problems require proving non-regularity of a given non-regular language.

The following examples illustrate the two problem classes.

**Example 1.** Construct a DFA \( A \) accepting the language \( L_1 \subseteq \{a, b\}^* \) containing words in which the number of occurrences of substring \( ab \) is equal to the number of occurrences of substring \( ba \).

**Example 2.** Prove the non-regularity of the language \( L_2 \subseteq \{a, b\}^* \) of the words with equal number of occurrences of \( a \)'s and \( b \)'s.

For the key components of an ITS listed above, we provide algorithms and techniques for the following: (a) automated checking of solutions (i.e., verification); (b) solution generation (i.e., synthesis); and (c) problem generation (relying on synthesis for generating interesting problems). We believe that these techniques can lead towards other features, such as fine-grained grading (as opposed to yes/no/counter-example style), and hint generation.

**Specification of automata problems.** We first address the challenge of formalizing the synthesis task that generates solutions to DFA construction and non-regularity proof problems. We provide languages for problem statements (specifications) and solutions.

The problems are stated in a **problem-description language** (PDL). Given an alphabet \( \Sigma \), a PDL expression \( \varphi \) defines a language \( L(\varphi) \) on \( \Sigma \). We require two properties of PDLs. The first is expressiveness: it should be possible for a human or a natural language processing tool to convert a natural language problem description to a PDL expression in a straightforward way. The second requirement concerns an algorithmic interface of a PDL to enable algorithmic problem solving. We have identified two such PDLs. We call the first the **user-friendly PDL** (UFPDL), which is a language that enables easy specification of a great majority of problems defined in natural language in courses and textbooks we
examined. The second is the context-free grammar PDL (CFPDL), which allows easy specification of problems given by context-free grammars and recursive definitions. Returning back to the two examples, in UFPDL, the language $L_1$ would be specified by the formula $\varphi_1 := \text{count}(ab) = \text{count}(ba)$ and the language $L_2$ by $\varphi_2 := \text{count}(a) = \text{count}(b)$, where $\text{count}(s)$ is the number of occurrences of $s$ in the string over which the formula is interpreted.

The solutions to automata construction problems and non-regularity proof problems are given as standard deterministic finite state machines and Myhill-Nerode style proofs, respectively. The non-regularity proof is represented as a pair consisting of a congruence function $F$ and a witness function $G$. The congruence function is a symbolic representation of the infinite family of equivalence classes, and the witness functions provide strings to certify that equivalence classes given by $F$ are distinct. More concretely, $G(i, j)$ returns a string that distinguishes between the equivalence classes represented by $F(i)$ and $F(j)$. We describe an expressive language $E_S$ for representing congruence and witness functions. As an example, for the language $L_2$, the congruence function is $F(i) = a^i$, and the witness function is $G(i, j) = b^i$.

Verification and synthesis for automata construction problems. For automata construction problems, the verification problem is: given a PDL expression $\varphi$ and a DFA $A$, verify that $L(A) = L(\varphi)$. The synthesis problem is: given a PDL expression $\varphi$ such that $L(\varphi)$ is regular, generate a DFA $A$ with $L(A) = L(\varphi)$.

We provide two algorithms for the verification problem. We assume that the PDL has a membership oracle, i.e. an algorithm for deciding whether a string $w$ belongs to $L(\varphi)$ for a PDL expression $\varphi$. This assumption holds for even very expressive PDLs. Our first algorithm works for any PDL (and uses its membership oracle), but requires that $L(\varphi)$ is regular (regularity assumption) and that the size of the smallest DFA for $L(\varphi)$ is bounded by a fixed constant (boundedness assumption). This assumption is justified as problems in homework assignments usually have solutions that are small (within 12 states, as our study shows). The algorithm carefully exploits a simple automata-theoretic theorem and the structural information in the automaton. Using this optimization enables our approach to scale to sizes typical for problems that occur in textbooks and homework assignments. The second algorithm is specific for UFPDL, and does not require the regularity and boundedness assumptions. Furthermore, it is much more efficient in practice than the generic algorithms. The algorithm reduces the verification problem for UFPDL to the model checking problem for the quantifier-free fragment of quantitative logic $\text{EF}^\Sigma^*$ [2]. We solve this problem through a novel reduction to QFBAPA satisfiability checking [9]. Our techniques also improve the best known complexity of (quantifier-)full $\text{EF}^\Sigma^*$ model checking from quadruply-exponential to triply-exponential\(^5\).

The skeleton of our automata synthesis algorithm is the classical L* methodology, which can efficiently learn an automaton for a regular language from a

\(^5\) We have since learnt through personal communication from the authors of [2] of a different, unpublished algorithm of triply-exponential complexity.
teacher who can answer membership and equivalence queries. The membership query corresponds precisely to the membership oracle of the PDL. The equivalence query asks for an equivalence between \( L(\varphi) \) and \( L(A) \) (where \( A \) is current guess of a solution). We can use one of the two verification algorithms mentioned above to answer the equivalence query.

 Verification and synthesis for non-regularity proof problems. For non-regularity proof problems, the verification problem statement is: given a PDL expression \( \varphi \), an expression in \( E_S \) representing a congruence function \( F \), and an expression in \( E_S \) representing a witness function \( G \), verify that \( F \) and \( G \) indeed are a Myhill-Nerode proof of non-regularity of \( \varphi \). The synthesis problem statement is: given a PDL expression \( \varphi \) such that \( L(\varphi) \) is non-regular, generate an expression in \( E_S \) representing a congruence function \( F \), and an expression in \( E_S \) representing a witness function \( G \).

For verifying \( F \) and \( G \) functions (given as \( E_S \) expressions), we prove a version of the Myhill-Nerode theorem that lets us to use the symbolic membership oracle for automatically verification the given witness and congruence functions. The symbolic membership oracle takes an expression \( S \in E_S \) and an expression \( \varphi \) in the PDL, and it checks whether all (or none) of the words in the language defined by \( S \) are in \( L(\varphi) \). In practice, do not require a complete algorithm, but only one that works in cases of interest. We provide such sound, but incomplete algorithms for both PDLs we consider.

For synthesizing the non-regularity proofs, we use the guess-verify methodology. First, the witness and congruence functions \( F \) and \( G \) are “guessed” using inductive synthesis techniques that rely on the membership oracle. The key idea for synthesizing proofs consists of two steps: (i) use a regular approximation of the given non-regular language to learn \( F \) for small values of \( i \), e.g. \( F(1) \) and \( F(2) \) (ii) use a novel inductive synthesis algorithm (of a similar style as proposed in [3]) to synthesize candidate \( F \) functions. The function \( G \) is synthesized using a similar idea.

Problem generation. We present an application of the synthesis algorithms to generating similar and dissimilar problems, for a given seed problem. The first step consists of generating variants from a given seed problem by replacing operators and operands by operators and operands of the same signature. The second step requires (a) judging whether a problem is well-formed, and whether its solution is trivial; and (b) classify the non-trivial problems into equivalence classes based on their similarity. It is here where our synthesis algorithms are especially useful. The failure of the synthesis algorithm is indicative of non well-defined problems, while much smaller solutions are indicative of trivial problems. Partitioning of problems into equivalence classes that indicate similarity can be performed by comparing features of the corresponding solutions generated by the synthesis algorithm. We remark that one such interesting feature for automata construction problems is to compare the automata structure while ignoring edge labels.
Experimental evaluation. We present detailed experimental results showing that our technique can synthesize solutions for most of the 100+ problems that we collected from different textbooks and courses on automata theory. For constructing automata, we had 108 different examples of which we could translate 94% into UFPDL. Using the equivalence check based on the membership query our tool constructs the correct DFA in 84% of these translated problems before reaching the 30 seconds timeout. Using the equivalence algorithm specific to the UFPDL we can solve 94% of the problems. In these examples, the median size of the learned automaton is 4 and 89% have 12 states or fewer. The median time for solving a problem is 2 seconds. Similarly, for proving non-regularity, we looked at 21 examples of which, 16 were expressible in either our UFPDL or CFPDL. We were able to automatically synthesize a non-regularity proof for each in less than 20 seconds each, and in 11 cases we needed less than 5 seconds. The congruence and witness functions were expressible in our language $E_S$ for each of these problems. Furthermore, the automated inductive theorem prover we constructed for the symbolic membership tests was found to be both sufficient and efficient for all the examples we had. For problem generation, the results show that for our set of problems, the approach can generate dozens of variants, both with isomorphic solutions and with differing solutions.

Summary. This paper makes the following contributions:

- We formalize the notions of verifying and synthesizing solutions for classical problems in automata theory. We introduce two languages for specification of problem statements that capture the great majority of problems occurring in practice. Also, we introduce a language for representing Myhill-Nerode proofs which is sufficient to express most of the proofs of non-regularity in our examples.
- We present algorithms for verifying an automaton and for synthesizing one, given an expression in any PDL that is equipped with a membership oracle. The verification algorithm builds over a non-trivial usage of a result from automata theory that bounds the size of a string that distinguishes between two equivalence classes defined by a regular language. Also, we show that equivalence problem of an expression in UFPDL and an automaton is decidable. The synthesis algorithm builds over the verification algorithm using the classical $L^*$ learning approach.
- We present algorithms for verifying a given proof of non-regularity and for synthesizing one, given a description of a non-regular language in any PDL that is equipped with a symbolic membership oracle. The verification algorithm uses a new alternate formulation of the Myhill-Nerode theorem and our implementation of the symbolic membership test for UFPDL. The synthesis algorithm builds over the verification algorithm using a non-trivial guess-and-verify loop.
- We describe how the synthesis algorithm can be used for generating similar and dissimilar problems, given a seed problem.
We present detailed experimental results illustrating the effectiveness of our techniques in specifying, synthesizing solutions to, and generating similar/dissimilar variants to 100+ problems collected from various courses and textbooks.

## 2 Specification of automata problems

In this section, we define our formalisms for problem specifications and their solutions. Identification of these formalisms, which are expressive enough to model a large number of real-world automata problems, but restricted enough to allow efficient algorithms, is one of the key contributions of this paper.

### 2.1 Solution Description Languages

In this section, we formalize our languages for describing solutions for automata construction and non-regularity proof problems.

**Automata.** An automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$, where $Q$ is a set of states, $\Sigma$ is a finite alphabet, $\delta$ is a transition function, $q_0$ is an initial state, and $F$ is a set of final states.

**Non-regularity Proofs.** A non-regularity proof is represented using a pair of two functions (referred to as congruence and witness functions, as will be explained in Section 4) that are parameterized by non-negative integer variables. We will use the following language $E_S$ for representing these functions:

$$S ::= t_1 v_1^1 t_2 v_2^2 \ldots t_n v_n^n$$

$$t ::= \text{str_const}$$

$$v ::= i | j | \text{int_const}$$

We refer to expressions in $E_S$ as symbolic strings. An example of a symbolic string is $(ab)^i a^j$. We have found language $E_S$ to be expressive enough for expressing the non-regularity proofs in the exercises we collected from textbooks and courses.

### 2.2 Problem description languages

A problem description language (PDL) allows describing a formal language, i.e., a PDL expression defines a set of words over an alphabet. Additionally, we require that two types of queries are decidable for a PDL: membership and symbolic membership.

More formally, a problem description language (PDL) is a tuple $(E, O_M, O_S)$, where:

- $E$ is a set of expressions (defined by a formal grammar). Given a finite alphabet $\Sigma$, an expression $\varphi$ in $E$ defines a set of words (that is, a language) $L(\varphi)$ over $\Sigma$. 

- $O_M$ is a membership oracle. Its input consists of an expression $\varphi \in E$, a finite alphabet $\Sigma$, and a word $w \in \Sigma^*$. The oracle $O_M$ outputs true iff $w \in L(\varphi)$, and outputs false otherwise.

- $O_S$ is a symbolic membership oracle. Its input consists of an expression $\varphi \in E$, a finite alphabet $\Sigma$, and an expression $S \in E_S$, where $E_S$ is the language used for representing non-regularity proofs as defined above in Section 2.1. Given a finite alphabet $\Sigma$, an expression $S$ in $E_S$ defines a set of words $L(S)$ over $\Sigma$. The oracle $O_S$ supports the following tests: (1) a positive symbolic membership test which is true iff $\forall w \in \Sigma^* : w \in L(S) \Rightarrow w \in L(\varphi)$, and (2) a negative symbolic membership test which is true iff $\forall w \in \Sigma^* : w \in L(S) \Leftrightarrow w \notin L(\varphi)$.

Note that the positive symbolic membership test given by symbolic membership oracle is a generalization of the membership oracle. The reason we choose to define the membership oracle and the symbolic membership oracle separately is that only membership oracle is needed for automata construction problems (whereas the symbolic membership oracle is needed for nonregularity proof problems). The above definition requires that the membership oracle and the symbolic membership oracle always output either true or false. We also consider sound, but incomplete oracles that might not terminate (or output “don’t know” due to timeout). In the remainder of this section, we define two PDLs.

User-friendly PDL. The goal of the first PDL we define is to cover naturally the languages mentioned in the textbooks and courses on regular automata. We therefore call the language the user-friendly PDL, or UFPDL. More technically, the language should be expressive enough to enable a translation from a natural language description to formula in a PDL using standard machine learning technology (for work on translating to a logic, see e.g. [17])

The language of UFPDL is defined by the grammar in Figure 1. A UFPDL formula is interpreted over strings over finite alphabet $\Sigma$. A formula $\varphi$ thus defines a set of strings, that is, a language. If we interpret a UFPDL formula $\varphi$ over a string $w$, the quantification over positions is interpreted as positions of $w$ and the quantification over strings is interpreted as quantifications over substrings of $w$. Note also that some functions in UFPDL that take an optional parameter, such as the function $\text{count}$. If we interpret a UFPDL formula $\varphi$ over a string (let the name of the string be $w$), and the optional parameter to a function is not provided, we take the parameter to be $w$. Thus $\text{count}(ab)$ counts occurrences of $ab$ in $w$, whereas $\text{count}(s,ab)$, where $s$ is a string variable, counts occurrences of $ab$ in the string $s$.

In the interest of space, we do not formally define the semantics of UFPDL. We illustrate the language expressions and motivate some of our design choices

---

6 We emphasize that we do not expect beginning students of automata theory to write formulas in UFPDL. They could use the natural language front-end (currently under development). Teachers can perhaps use UFPDL directly, and new interesting problems are generated by automatically modifying UFPDL expressions (see Section 5).
pos ::= pos_var | pos_const |
| indexOf(str_const) | next(pos) | prev(pos)

int ::= int_const | card(set) | intofpos(pos)
| count(str?,str_const) | length(str?)
| int + int | int - int | int mod const

char ::= char_const | fromEnd(str?,pos_const)
| fromStart(str?,pos_const) | atPos(pos)

str ::= str_var | str_const | substr(pos,pos) | char

set ::= indexesOf(str_const) | symbols(str?)
| set_const | set ∪ set | set ∩ set | set \ set

cint ::= int_const | int_var | cint + cint
| cint - cint | cint mod const

fmt ::= str_const | str_const \ intvar | fmt.fmt

constr ::= cint < cint | cint = cint | ¬ constr
| constr ∧ constr | constr ⇒ constr

reg ::= = ε | char_const | reg + reg | reg.reg | reg^*

predicate ::= pos = pos | int = int | char = char
| set = set | int < int | isSuffix(str?,str_const)
| char ∈ set | regex(reg,str?) | str = str_const
| isPrefix(str?,str_const) | pos ∈ set | set ⊆ set
| contains(str?,str_const) | match(fmt,constr,str?)

ϕ₀ ::= predicate | ϕ₀ ∧ ϕ₀ | ¬ ϕ₀ | ϕ₀ ⇒ ϕ₀

ϕ₃ ::= ϕ₀ | ∃posvar. ϕ₃ | ∃strvar. ϕ₃

ϕᵥ ::= ϕ₀ | ∀posvar. ϕᵥ | ∀strvar. ϕᵥ

ϕ ::= ϕ₃ | ϕᵥ

Fig. 1: Grammar of user-friendly problem description language
using the following examples. All examples come from existing textbooks and courses, the full list of examples and their sources is presented in the appendix.

**Example 3.** The language of strings that have the symbol \( a \) on all odd positions can be expressed in UFPDL as follows: \( \forall p : (\text{intofpos}(p) \mod 2 = 1) \rightarrow \text{atPos}(p) = a \). This example demonstrates some of the operations on positions. Note the conversion of positions to integers for mathematical operations. The distinction between integers and positions prevents some operations on positions, for example for positions \( p_1, p_2 \), one cannot access a string at position \( p_1 + p_2 \).

**Example 4.** Consider the language of strings consisting of one or more \( a \)'s followed by one or more \( b \)'s. The corresponding formula in UFPDL is \( \text{match}(a^n\, b^m, n > 0 \land m > 0) \). This example illustrates the \text{match} construct, which takes two (optionally three) parameters: a format string, a constraint on the integer variable mentioned in the format string, and (optionally) a string variable that is matched with the format string. If the last parameter is not provided, we interpret the \text{match} predicate over the model of the formula. For example, the string \( aaabbbb \) is in the language defined by the expression, with \( n \) instantiated to 3 and \( m \) to 4. The \text{match} predicate provides a limited replacement for quantification over integers. We do not allow general quantification over integers in order to keep the membership test decidable with a low complexity. For the same reason, we allow only restricted number of operation on integers occurring in the match predicate. These operations are defined by the \text{cint} production rule in the UFPDL.

**Example 5.** The set of strings having \( n \) \( a \)'s followed by \( n \) \( b \)'s, i.e., \( \text{match}(a^n\, b^n, n = m) \) is a non-regular set expressible in UFPDL.

**Example 6.** The language of strings where all occurrences of 01 start on an even position is equivalent to the UFPDL formula \( \forall p : \text{substr}(p, p + 2) = "01" \Rightarrow (p \mod 2) = 0 \). This illustrates the \text{substr}(\( p_1 \), \( p_2 \)) function which returns the substring between the positions \( p_1 \) and \( p_2 \).

**Example 7.** The language of words where all non-zero even length prefixes end with \( a \) can be used to demonstrate quantification over substrings: the formula we obtain is \( \forall v. (\text{isPrefix}(v) \land \text{length}(v) > 0 \land ((\text{length}(v) \mod 2) = 0)) \rightarrow \text{fromEnd}(v, 1) = a \). Although it can be defined using simpler constructs, this formulation demonstrates that the UFPDL is rich enough to capture the natural language statement directly. This statement was taken from an existing course, see the appendix for its source.

To complete the definition of the PDL, we need an algorithm for the oracles. The membership oracle tests the membership of a given string \( w \) in a UFPDL expression \( \varphi \). Giving an algorithm for the membership oracle is straightforward, and we omit it due to space constraints. A sound, but incomplete algorithm implementing the symbolic membership oracle will be described in Section 4.

We emphasize that our goal in designing UFPDL was not to extend it to cover all problems, but to have a sufficiently high-level language that captures a great majority of examples occurring in practice, and at the same time preserves
the possibility for efficient algorithms (see Section 3 and 4). This is the reason why we do not allow quantifier alternation or more complex integer operations in UFPDL. For an example of a problem that cannot be captured in UFPDL, see Benchmark 55 in the appendix. The benchmark requires reasoning about encoding of integers in binary numbers, which is not supported in UFPDL.

**Context Free Grammar PDL.** The next PDL we present is called a context-free PDL (or CFPDL), and is given by a context-free grammar. In what follows, the non-terminals will be denoted by \( N, N_1, \ldots \), and the rules will be of the form \( N \to \sigma \), where \( \sigma \) ranges over words in \( V \cup \Sigma \), where \( V \) is the set of non-terminals, and \( \Sigma \) is a finite alphabet. CFPDL is a natural PDL choice when students move on from learning regular languages to learning context-free language in a typical course on theory of computation. A standard category of problems related to this transition is about proving that a certain CFG (like set of palindromes or balanced parentheses) is non-regular, or showing that a language, with a possibly succinct CFG description, might still be regular.

The membership oracle solves the standard parsing of context-free grammars. A sound, but incomplete, algorithm implementing the symbolic membership oracle will be described in Section 4.

### 3 Automata Construction Problems

#### 3.1 Verification

For automata construction problems, the verification problem statement is: given a PDL expression \( \varphi \) and a DFA \( A \), verify that \( L(A) = L(\varphi) \). We provide two algorithms for this problem.

- The first one is for generic PDL, and uses only the membership oracle. It requires knowing a bound on the size of the automata to be synthesized. The key contributions in this algorithm are the careful exploitation of an automata-theoretic result (Proposition 1) and of the structural information in \( A \).
- The second is specific to UFPDL we have defined. It exploits the properties of constructs in UFPDL and is based on an improved algorithm for model checking the \( EF^\Sigma \) logic. (Note that due to a standard result in formal language theory, the verification problem is not decidable for CFPDL. For CFPDL, we thus need to use the algorithm mentioned above that works for a generic PDL.)

**Verification for a general PDL** The algorithm for verification for a general PDL will rely on the following two assumptions on \( L(\varphi) \): *Regularity*, i.e. \( L(\varphi) \) defines a regular language, and *Boundedness*, i.e. the size of the smallest DFA for \( L(\varphi) \) is bounded by a constant \( k \).
Equivalence algorithm 0. Given the boundedness assumption, an algorithm to check equivalence of a PDL formula and an automaton follows directly from a standard result in automata theory.

Theorem 1 (Theorem 3.10.5 in [13]). Let $A$ and $B$ be DFA with $k_1$ and $k_2$ states respectively with $L(A) \neq L(B)$. There is a word $w$ in $(L(A) \setminus L(B)) \cup (L(B) \setminus L(A))$ with length at most $k_1 + k_2 - 2$.

The idea on which the algorithm is based is simple: if $Q_A$ is of size $k_A$ and the minimal DFA for $\varphi$ has $k$ states (this is due to the assumptions), then the shortest counterexample to equivalence can be of length at most $k_A + k - 2$. This is due to Theorem 1. The algorithm thus tests whether $\varphi$ and $A$ agree on all strings with lengths smaller or equal to $k_A + k - 2$. In the algorithm below, we set the bound $B$ to $k - 2$. The running time is $O(|\Sigma||Q_A| + B)$.

Algorithm 0

Equivalence Algorithm 0

Input: PDL Expression $\varphi$, Automaton $A$, bound $B$

Output: true (equivalent); false (not equivalent) + counterexample

int $k_A \leftarrow$ number-of-states($A$)

for all string $s \in \bigcup_{0 \leq i \leq k_A + B} \Sigma^i$ do

    if $\neg(s \in L(A) \Leftrightarrow s \in L(\varphi))$ then return false, $s$

return true

Equivalence algorithm 1. If we use the structure in $A$, we can get an algorithm with significantly improved performance (both in terms of computational complexity and empirical behavior). We use the following result (Proposition 1). It is stated in terms of the Myhill-Nerode equivalence relation, and its proof is similar to the proof of Theorem 1 (the proof is in the appendix.)

Let $L$ be a regular language over alphabet $\Sigma$. It defines the equivalence relation $\equiv_L$ on strings over $\Sigma$ as follows: for all strings $u$ and $v$ in $\Sigma^*$, we have $u \equiv_L v$ iff $\forall w \in \Sigma^* : uw \in L \iff vw \in L$. Similarly, we define the following family of equivalence relations $\equiv_k^L$: for all strings $u$ and $v$ in $\Sigma^*$, we have $u \equiv_k^L v$ iff $\forall w \in \Sigma^k : uw \in L \iff vw \in L$.

Proposition 1. Let $A$ be the minimal DFA for a regular $L \subseteq \Sigma^*$, and let $k = |A|$. For all $u, v \in \Sigma^*$, $u \equiv_k^L v \implies u \equiv_L v$.

Algorithm 1 relies on Proposition 1, and on the regularity and boundedness assumptions. We set the bound $B$ for the algorithm to $k - 2$, where $k$ comes from the boundedness assumption.

To understand Algorithm 1, consider a tree $T$ representing all the strings in $\Sigma^*$: the root is labeled by an empty string, each node has $|\Sigma|$ successors, and if
Algorithm 1 Equivalence Algorithm 1

Input: PDL Expression $\varphi$, Automaton $A$, bound $B$
Output: true (equivalent); false (not equivalent) + counterexample

1: enqueue($\epsilon, q_0, D$); $U \leftarrow \emptyset$
2: while nonempty($D$) do
3: $(s, q) \leftarrow$ dequeue($D$); add($q, U$);
4: for all $a \in \Sigma$ do
5: $s' \leftarrow sa; q' \leftarrow \delta_A(q, a)$;
6: if $q' \in U$ then
7: $(b, ctrex) \leftarrow$ check($s', q', \varphi, B$)
8: if $¬b$ then return false, ctrex
9: else enqueue($\epsilon, q_0, D$);
10: return true

In Algorithm 1, we use a queue $D$ to represent a breadth-first search of the tree $T$. Each element in the queue is a label of a node in $T$, i.e., a tuple $(s, q)$, where $s$ is a string and $q$ is a state of $A$. In line 1, $D$ is initialized by enqueueing the label of the root of $T$, i.e., the pair $(\epsilon, q_0)$, where $\epsilon$ is the empty string and $q_0$ is the initial state of $A$. We also maintain a set $U$ of states of automata $A$ that have been processed. The while loop processes elements in the queue. After dequeuing a pair $(s, q)$ (line 3), the state $q$ is added to states that were already seen (line 3). After that, for each successor $(s', q')$, we check whether the state $q'$ was already in $U$. If not, we enqueue the pair $(s', q')$ and continue.

The key part of the algorithm is in lines 7-8. There we are in the case when we are processing a node of the tree $T$ labeled with a state that was already seen. The execution has arrived to the state $q'$ on prefix $s'$. Suppose the first time the execution arrived to $q'$ was on prefix $s_p$. We therefore know that the automaton $A$ does not distinguish between $s'$ and $s_p$ as prefixes. If $\varphi$ and $A$ are equivalent the same must hold for the expression $\varphi$, i.e., the following property $\psi$ must hold for $\varphi$: for all $w$, a word $s'w$ is in $L(\varphi)$ iff a word $s_pw$ is in $L(\varphi)$. We check $\psi$ using Proposition 1. The proposition asserts that if we check the property $\psi$ for all $w$ of length less than $k-2$ (the number $k$ comes from the Boundedness assumption on $L(\varphi)$), the property $\psi$ must hold for all $w$. The check procedure (called in line 7) checks, for all $w$ of length less than $B$ (recall $B$...
was initialized to \(k - 2\), that the automaton \(A\) and the formula \(\varphi\) agree on \(s'w\). If they do, then we can conclude that \(\psi\) holds, as a similar check was in effect performed on \(s_p\) when it was encountered. If they do not agree, the algorithm has found a counterexample, and returns it (line 8). It is possible to prove that if the algorithm terminates without finding such a counterexample, \(A\) and \(\varphi\) are equivalent.

The complexity of the algorithm is derived as follows. There are at most \(|Q|\) calls to enqueue (where \(Q\) is the set of states of \(A\)). Each of the enqueued tuples will be dequeued before the algorithm finishes, which will cause at most \(|\Sigma|\) calls to the check procedure. The check procedure checks \(|\Sigma|^B\) strings, and its complexity is therefore \(O(|\Sigma|B)\). The overall complexity of the algorithm thus is \(O(|Q||\Sigma|B+1)\). We obtain the following theorem.

**Theorem 2 (Equivalence Algorithm Correctness).** Let \(\varphi\) be a PDL formula for which the Regularity and Boundedness assumptions hold. Let \(A\) be a finite state automaton. Equivalence Algorithm 1, given inputs \(\varphi\) and \(A\), outputs true iff \(L(A) = L(\varphi)\), and outputs false and a string \(\text{ctrex}\) iff \(L(A) \neq L(\varphi)\). In this case \(\text{ctrex}\) is in \((L(A) \setminus L(\varphi)) \cup (L(\varphi) \setminus L(A))\).

**Verification for UFPDL** We present an algorithm for the verification problem for UFPDL that is (a) empirically much more efficient than the generic algorithms; and (b) does not need the regularity and boundedness assumptions.

**Monitor Automata.** A monitor automaton \(M = \langle Q, \Sigma, \delta, q_0, F(\phi_i, (L_{\phi_i})) \rangle\) where \(Q, \Sigma, \delta, q_0, \) and \(F\) are as in a standard finite automaton, and \(\langle \phi_i \rangle\) is a tuple of quantifier-free UFPDL terms and \((L_{\phi_i})\) are corresponding annotation functions.

Each annotation function \(L_{\phi_i}\) labels transitions of \(M\), and can be used to compute the value of term \(\phi_i\) along any run of \(M\). Firstly, let \(w = \sigma_0\sigma_1 \ldots \sigma_{n-1} \in \Sigma^*\) and let \(q_0q_1 \ldots q_n\) be the run of \(M\) on \(w\). Each annotation function \(L_i: Q \times \Sigma \rightarrow \mathcal{V}_i\) is of one of the following forms:

- **Value annotation.** Type of \(\mathcal{V}_i\) is the type of the expression \(\phi_i\), and \(\phi_i(w)\) is equal to \(L_{\phi_i}(q_{n-1}, \sigma_{n-1})\). Intuitively, the value of \(\phi_i(w)\) is the annotation of last transition in the run.
- **Sum annotation.** Type of \(\mathcal{V}_i\) and \(\phi_i\) is \(\text{int}\), and the value \(\phi_i(w)\) is equal to the sum \(\sum_{k=0}^{n-1} L_{\phi_i}(q_k, \sigma_k, q_{k+1})\), i.e., the value of \(\phi_i(w)\) is the sum of all annotations along the run.
- **Indicator annotation.** We have \(\mathcal{V}_i = \{0,1\}\) and type of \(\phi_i\) is either \(\text{pos, str}\) or \(\text{set(pos)}\). We further impose the condition that the annotation is 1 only along one transition of a run if \(\phi_i\) is of type \(\text{pos}\), and that it is 1 only along a contiguous block of transitions in a run if \(\phi_i\) is of type \(\text{str}\). Let \(K = \{k \mid L_{\phi_i}(q_k, \sigma_k, q_{k+1}) = 1\}\). We have that the value \(\phi_i(w)\) is (a) the unique \(k \in K\) if \(\phi_i\) is of type \(\text{pos}\); (b) \(K\) if \(\phi_i\) is of type \(\text{set(pos)}\); and (c) \(\sigma_k\sigma_{k+1} \ldots \sigma_{k+l}\) where \(\{k, k+1, \ldots, k+l\} = K\) if \(\phi_i\) is of type \(\text{str}\).

Intuitively, annotation functions denote how the value of a term is to be updated with each additional symbol in a string. We denote by \([M, w, \phi]\), the value of
ϕ(\(w\)) computed according to the annotation function \(L_\phi\) on the run of \(w\) in \(M\). Furthermore, we abuse notation by calling \(L_\phi(q,\sigma)\) the annotation of the transition \((q,\sigma,\delta(q,\sigma))\) by \(L_\phi\).

**Example 8.** (Illustrative example) Consider the following UFPDL formula: \(\varphi := \text{count}(ab) = \text{count}(ba) \equiv \text{count}(ab) - \text{count}(ba) = 0\) and a monitor automaton \(M\) in Figure 2. Let \(\phi_1, \phi_2\) and \(\phi_3\) denote \(\text{count}(ab), \text{count}(ba)\) and \(\text{count}(ab) - \text{count}(ba)\), respectively. In the figure, the first, second, and third component of the label of each transition in \(M\) correspond to \(L_{\phi_1}, L_{\phi_2}\), and \(L_{\phi_3}\), respectively. Each of \(L_{\phi_1}, L_{\phi_2}\), and \(L_{\phi_3}\) is a sum annotation function, i.e., along the run in \(M\) of any word \(w \in \{a, b\}^*\), we have that:

- The sum of the first (resp. second) component is \(\text{count}(ab)\) (resp. \(\text{count}(ba)\)) as every occurrence of \(ab\) takes either \(1 \to 3\) or \(2 \to 4\) (resp. \(3 \to 1\) or \(4 \to 2\)) which adds 1 to the sum.
- The sum of the third component is \(\text{count}(ab) - \text{count}(ba)\) as for every transition, \(L_{\phi_3}\) (third component) is the difference between \(L_{\phi_1}\) (first component) and \(L_{\phi_2}\) (second component).

We check whether the language of \(M\) is equivalent to \(\varphi\) by testing for witness counter-example runs of the following form: (a) end in an accepting state of \(M\), but have the sum of the weights in the third component is not zero, or (b) end in an non-accepting state of \(M\), but have the sum of the weights in the third component equal to zero. We have that the language of \(M\) is equivalent to \(\varphi\) if and only if there are no witness counter-example runs.

The decidability result for checking the equivalence of a UFPDL formula and an automaton is based on two observations: (a) For any sentence \(\psi\) of UFPDL and automat \(A\), a monitor automaton \(M(\psi, A)\), which contains an annotation function for every atomic term in \(\psi\), can be effectively constructed; and (b) Finding witness counter-example runs in \(M(\psi, A)\) is decidable.

**Algorithm.** Algorithm 2 for checking equivalence of UFPDL formula \(\Phi\) and \(A\) is based on: (a) constructing monitor \(M\) containing \(L_\psi\) for every sub-term \(\psi\) of \(\Phi\); and (b) finding witness counter-example runs in \(M\) is decidable. Let \(\Phi = \forall p_1, \ldots, w_1, \ldots : \Phi_{QF}\) with type of \(p_i\)’s and \(w_i\)’s being position and substring, respectively. Existentially quantified \(\Phi\) are handled by first complementing both \(A\) and \(\Phi\). Assume that \text{match} does not occur in \(\Phi_{QF}\) (see the appendix on
handling \texttt{match}). The key steps of the algorithm are: (1) augment \( \mathcal{A} \) to \( \hat{\mathcal{A}} \) to guess values for \( p_i \)'s and \( w_i \)'s; (2) build monitor \( \mathcal{M} \) from \( \hat{\mathcal{A}} \) having annotations for every subterm of \( \Phi_{QF} \); and (3) check for witness counter-example runs in \( \mathcal{M} \).

\begin{algorithm}
\textbf{Algorithm 2} UFPDL Equivalence Algorithm

\textbf{Input}: Automaton \( \mathcal{A} \), UFPDL formula \( \Phi = \forall \forall \exists : \Phi_{QF} \)

\textbf{Output}: true (equivalent); false (not equivalent) + counterexample

1: \( \hat{\mathcal{A}} \leftarrow \text{GuessVariables}(\mathcal{A}, \forall \exists \in \Phi) \); \( \mathcal{M} \leftarrow \text{AddAnnotation}(\hat{\mathcal{A}}, \forall \exists) \)
2: \textbf{while} \( \exists \phi \) sub-expression of \( \Phi_{QF} \); sub-expressions of \( \phi \) are annotated and \( \phi \)

\hspace{1cm} is not an integer comparison \textbf{do}
3: \( \mathcal{M} \leftarrow \text{AddAnnotation}(\mathcal{M}, \phi) \)
4: \( \text{cex} \leftarrow \text{FindCounterExamplePath}(\mathcal{M}, \Phi_{QF}) \)
5: \textbf{if} \( \text{cex} = \bot \) \textbf{return} true \textbf{else} return false, cex
6: \textbf{return} true

\end{algorithm}

\textit{Guessing the quantified variables}. We build \( \hat{\mathcal{A}} \) on \( \hat{\Sigma} = \Sigma \times 2^{\{p_1, \ldots, p_l\}} \) to guess values for \( p_i \)'s and \( w_i \)'s. Consider the following conditions on \( (q_0, V_0) \ldots (q_n, V_n) \in \hat{\Sigma}^* \): (a) \( q_0 \ldots q_n \in L(\mathcal{A}) \); (b) for all \( p_i, p_i \in V_j \) for only one \( j \); and (c) for all \( w_i, w_i \in V_j \) for only one contiguous sequence of \( j \)'s. Let \( F_w \) and \( F \) be two separate accepting sets for \( \mathcal{A} \) such that: \( F_w \) accepts words where (b) and (c) hold (well-formed guesses for variables); and \( F \) accepts words where (a), (b) and (c) hold.

\textit{Construction of } \mathcal{M}. We construct the monitor automaton \( \mathcal{M} \) from \( \hat{\mathcal{A}} \) inductively. First, for every quantified variable \( v \), we add an (indicator) annotation function \( \mathcal{L}_v \) such that \( \mathcal{L}_v(q, \sigma, V) \) is 1 if and only if \( v \in V \). Then, if all subterms of \( \phi \) are annotated, we add \( \mathcal{L}_\phi \) to \( \mathcal{M} \) (unless \( \phi \) is an integer comparison).

The key property is that UFPDL functions are \textit{finite-memory updatable}, i.e., change between \( \phi(w) \) and \( \phi(w \cdot \sigma) \) depends only on (a) change between \( \psi(w) \) and \( \psi(w \cdot \sigma) \) for subterms \( \psi \) of \( \phi \); and (b) finite information about \( w \).

\textbf{Example 1}. Let \( \psi = \text{indexesOf}(bab) \) and \( \phi = \text{count}(\psi) \). The value \( \Delta \phi = \phi(w \cdot b) - \phi(w) \) is computable from \( \Delta \psi = \psi(w \cdot b) \setminus \psi(w) \) (i.e., \( \Delta \phi = 1 \Leftrightarrow \Delta \psi \neq \emptyset \)). In turn, \( \Delta \psi \) depends only on finite information about \( w \) (i.e., does \( w \) end with \( ba? \) ). To extend \( \mathcal{M} \) with \( \mathcal{L}_\psi \), we take the product \( \mathcal{M} \times \mathcal{A}_{bab} \) (where \( \mathcal{A}_{bab} \) accepts words ending with \( bab \)) and let \( \mathcal{L}_\psi \) be 1 only for transitions leading to an accepting state in \( \mathcal{A}_{bab} \). Then, \( \mathcal{L}_\phi \) is a sum annotation with \( \mathcal{L}_\phi (s, \sigma) = \mathcal{L}_\psi (s, \sigma) \).

Here, we illustrate the procedure to inductively add an annotation function for \( \phi \) for a few constructions of the UFPDL. The rest of the constructs can be handled similarly.

\textbf{indexesOf}(str, const). Let \( \mathcal{A}' \) be an automaton that accepts words that end with \texttt{str,const}. We take the product of \( \mathcal{M} \) with \( \mathcal{A}' \), and let \( \mathcal{L}_\phi \) be an indicator annotation that assigns 1 to only those transitions that lead to final states in \( \mathcal{A}' \).
We state without proof that these conditions are sufficient to encode a path. Let $L$ and sets $X_i$ and having the boolean operations over sets and Presburger reasoning about their cardinalities.

$v$ is visited in $improve the complexity for full EF exponential in the size of the graph. We provide an improved algorithm for expression in the quantifier-free fragment of the quantitative logic EF and (b) false if it ends in $F_w \setminus F$ ($\tilde{w}$ is well-formed, but rejected). The runs which violate (a) (resp. (b)) are positive (resp. negative) counter-examples. We explain how to find negative counter-examples in $M$. The positive case is analogous.

Let all boolean subterms of $\Phi_{QF}$ be integer comparisons Any other boolean term $\phi$ is replaced by $v_\phi > 0$, where $v_\phi$ is a new $int$ variable which is incremented (resp. decremented) whenever $\phi$ changes from false to true (resp. true to false). Let $\phi_0, \ldots, \phi_n$ be the $int$ subterms of the boolean terms. We consider $M$ to be a multi-weighted graph with the states $Q$ as vertices, edges $E = \{(q, \sigma, \delta(q, \sigma))\}$, and having the $i^{th}$ weight component of $e \in E$ as $L_{\phi_i}(e)$ (here $L_{\phi_i}(e) = L_{\phi_i}(q, \sigma, \delta)$ where $e = (q, \sigma, q')$)

Let $\pi$ be a run of $w$ in $M$ in which each $e \in E$ occurs $x_e$ times. We have that $[[M, \phi_i', w]] = \sum_{e \in E}x_e \cdot L_{\phi_i'}(x_e)$, and hence, that $\pi$ is a negative counter-example iff $start(\pi) \in \{q_i\} \land end(\pi) \in F_w \setminus F \land \Phi_{QF}[\forall \phi_i : \phi_i/\sum_{e \in E}L_{\phi_i}(e) \cdot x_e]$. This is an expression in the quantifier-free fragment of the quantitative logic $EF^2$ [2]. EF$^2$ allows Presburger expressions over the sum of the weight components of the edges along a path. The model-checking algorithm for $EF^2$ from [2] was quadruply exponential in the size of the graph. We provide an improved algorithm for the quantifier-free fragment via reduction to QFBA [9] and then show how to improve the complexity for full $EF^2$. The logic QFBA allows reasoning about boolean operations over sets and Presburger reasoning about their cardinalities.

We encode the path $\pi$ in QFBA using non-negative integers $x_e$ (for $e \in E$) and sets $X_v$ (for $v \in Q$). Intuitively, edge $e$ occurs $x_e$ times in $\pi$ and $|X_v| = 1$ iff $v$ is visited in $\pi$. However, any valuation of $x_e$’s and $X_v$’s need not correspond to a real path in the graph; further conditions are needed on $x_e$’s and $X_v$’s. For vertex $v$, let $out(v) = \sum_{e = (v', \sigma, v)} x_e$ and $in(v) = \sum_{e = (v, \sigma, v')} x_e$ be the number of times edges leading to and leading from $v$ are taken.

- The sets $X_v$’s are disjoint, i.e., $\Psi_0 \equiv \bigwedge_{v \neq v'} X_v \cap X_{v'} = \emptyset$; and $v$ is visited iff $in(v)$ or $out(v)$ is non-zero, i.e., $\Psi_1 \equiv \bigwedge_v |X_v| = 1 \iff (in(v) + out(v) \neq 0)

- A vertex $v$ is entered the same number of times it is exited unless it is initial or final. The initial vertex $v_i$ is exited one more time than entered unless it is also the final vertex. The opposite holds for the final vertex. Formally, $\Psi_2 \equiv \bigwedge_{v \in Q} \sum_{e \in F_v \setminus F} \min(v) = out(v) \land 0 \leq out(v) - in(v) \leq 1 \land \bigwedge_{v \in F_v \setminus F} -1 \leq out(v) - in(v) \leq 0 \land \sum_{v \in F_v \setminus F} out(v) - in(v) = 0
\n- If $v$ is visited, a path $e_0 \ldots e_k$ from $v_i$ to $v$ exists with $\forall e_i : x_{e_i} > 0$. We encode this using additional sets $C_0$ to $C_{|M|}$ where $X_v \subseteq C_i$ iff there is a path to $v$ of length at most $i$. Furthermore, $\forall v : X_v \subseteq C_{|M|}$ as the path length can be bounded by the number of vertices. Formally, $\Psi_3 \equiv C_0 = X_{v_i} \land \bigcup_s X_v = C_{|M|} \land (\bigwedge_{0 \leq i < |M|} \bigwedge_v X_v \subseteq C_{i+1} \setminus C_i \implies \forall \nu(v : X_{v'} \subseteq C_i \bigvee x_{e,v,v'} x_e > 0))$

We state without proof that these conditions are sufficient to encode a path.
Lemma 1. Let $x_e$’s and $X_v$’s be such that $\Psi_0$ through $\Psi_3$ are satisfied. Then, there is a path $\pi$ in the monitor such that each edge $e$ occurs $x_e$ times in $\pi$.

Now, a negative counterexample is equivalent to a satisfying assignment $\Psi = \bigwedge_{k=0}^3 \Psi_k \land \Phi_{QF} \left( \forall i : \phi_i / \sum_{e \in E} x_e \cdot L_{\phi_i}(e) \right)$. The length of $\Psi$ is polynomial and exponential in $\mathcal{A}$ and $\Phi$, respectively, as $\mathcal{M}$ is polynomial in $\mathcal{A}$ and exponential in $\Psi$. As satisfiability of QFBAPA is NP-complete [9], the theorem follows.

Theorem 3. Checking equivalence of UFPDL formula $\Phi$ and automaton $\mathcal{A}$ is decidable in time doubly- and singly-exponential in sizes of $\phi$ and $\mathcal{A}$, respectively.

Application to full EF$^\Sigma$. In [2], a similar approach is taken to (quantifier-)full EF$^\Sigma$ model-checking, i.e., encoding in Presburger arithmetic the number $x_e$ of times edge $e$ appears in a path. However, the quadruply-exponential complexity arises as it is not easy to encode that $x_e$’s correspond to a valid path. Our use of sets $X_v$’s and $C_i$’s enables us to encode this condition in a BAPA formula. As BAPA satisfiability checking is equivalent to Presburger arithmetic, we get a triply-exponential algorithm for full EF$^\Sigma$.

3.2 Synthesis

The synthesis algorithm takes a PDL expression $\phi$ for which the Regularity and Boundedness assumptions hold, and returns an automaton $\mathcal{A}$ with $L(\mathcal{A}) = L(\phi)$. (If $\phi$ is in UFPDL then the Boundedness assumption is not needed.)

The framework of our approach is the classical $L^*$ algorithm [1, 11], which uses two types of queries to learn an automaton for an unknown regular language: a membership query and an equivalence query. The membership query corresponds precisely to the membership oracle of a PDL. On an equivalence query, the learner asks whether the automaton it has constructed is equivalent to the unknown language. The teacher replies positively or provides a counterexample. The equivalence query can be implemented by the verification algorithms in Section 3.1. The full description of $L^*$ can be found in the appendix.

3.3 Experimental Evaluation

We have implemented a system that takes as input a formula in the UFPDL, and the alphabet over which the formula is evaluated, and it outputs the corresponding minimal DFA. In particular, we have implemented the $L^*$ algorithm, the membership oracle for the UFPDL, and the equivalence check using each of the three methods: Algorithm 0, Algorithm 1, and Algorithm 2. We have evaluated our system on a set of 108 real benchmarks collected from undergraduate automata theory courses. During the evaluation we focus on checking whether (a) our UFPDL is expressive enough to encode most problems that fit the classic problem template of DFA construction, (b) the number of states in the DFAs for the benchmarks is usually small (less than 12), (c) our algorithm runs efficiently in practice, even though it is worst-case exponential in the bound on the number of states.
Expressiveness of UFPDL. Out of the 108 collected examples, 102 are expressible in our UFPDL corresponding to a success rate of 94%. The complete list of these problems with statements and corresponding formula can be found in the appendix. This confirms that UFPDL is expressive enough to capture most textbook exercises. The inexpressible problems include, for instance, problems that reason about the integer value of an binary input string and second order problems where the solution is a construction that takes parameters and returns an automaton.

Out of the 102 problems turned into the UFPDL, 15% of the problems were duplicates. We removed them as they do not provide any additional information and obtained 91 different problems.

Size of the Solutions. One of our starting hypothesis is that automata solutions in the domain of education are typically small. We need to confirm this assumption and to find a good value for the bound. Out of the 91 examples, 81 have 12 states or fewer (89%) and the median number of states is 4. This confirms the assumption that the automata are small. Figure 3 shows the distribution of the automaton sizes. For the sake of clarity, 4 automata are excluded: with sizes 32, 32, 1024, and 2047. We discuss in the appendix how, in some cases, we can learn automaton of such sizes. The second problem is to find a good value for the bound on the automata size. Algorithm 1 is exponential in this bound, so it has an important impact on the running times. We ran our algorithm with different values for the bound and found that 12 is a good accuracy vs time trade-off, where our implementation was able to solve all but one problem when we do not put a timeout. Reducing the bound quickly leads to a greater number of erroneous solutions. As mentioned, out of the 91 examples, 81 have 12 states or fewer (89%). However, for all but one example, the algorithms for generic PDLs were able to learn the correct solution (the algorithm specific to UFPDL learned correct solution for this problem). Next, we report results on these 90 problems.

Running Times. We now evaluate the practicality of our approaches. The implementation is done in the Scala programming language. All tests were run on a machine with an AMD Opteron 2431 CPU. We set a timeout of 30 seconds for the total time spent in the UFPDL solver, which dominates the total synthesis time. When the timeout is reached, the current guess is returned without any guarantee about the correctness of the solution.

The median time for learning an automaton is 2 seconds, but there is a large variation across examples. Figure 4 show the running time for the three methods. Even though the automaton can be learned with a small number of examples, we need to test it on many more to be certain that it is correct. Algorithm 0 was able to finish in 68% of the cases (63 examples), Algorithm 1 in 84% (76 examples). This shows the benefit of using Algorithm 1 over Algorithm 0. Algorithm 2 solves 85 examples (94%) and is performing around two orders of magnitude faster. Thus, it is reasonable to ask why do we keep Algorithm 1. It has the advantage of being general as it works for any PDL with a membership oracle and hence allows for any extensions to UFPDL.
4 Non-regularity Proof Problems

In this section, we present techniques for verification and synthesis of proofs of non-regularity.

4.1 Proof Framework

There are two methods used in a typical curriculum for establishing non-regularity of a language: the Myhill-Nerode Theorem and the pumping lemma. We describe a technique for automating the former method since it is also a necessary condition. The Myhill-Nerode theorem is a classical characterization of regular languages in terms the relation $\equiv_L$ defined in Section 3.

**Theorem 4 (Myhill-Nerode Theorem).** A language $L$ is regular iff $\equiv_L$ is of finite index. Equivalently, $L$ is non-regular iff there exists functions $F : \mathbb{N} \rightarrow \Sigma^*$ and $G : \mathbb{N} \times \mathbb{N} \rightarrow \Sigma^*$ such that:

$$\forall 1 \leq i < j : F(i)G(i,j) \in L \iff F(j)G(i,j) \notin L \quad (1)$$

We refer to $F$ and $G$ as congruence functions, and witness functions respectively. For a non-regular language, the congruence function represents an infinite family of $\equiv_L$ congruence classes (with representatives $F(i) : i \in \mathbb{N}$), with $G(i,j)$ witnessing $F(i) \not\equiv_L F(j)$. Hence, a Myhill-Nerode proof of non-regularity is a congruence function $F$ along a witness function $G$. Henceforth, we use congruence
and witness functions expressible in the $E_S$ (Section 2) and for our benchmarks, this turns out to be sufficient.

We now state an equivalent form of Myhill-Nerode Theorem more amenable to automation.

**Theorem 5 (Alternate Form of Myhill-Nerode Theorem).** The condition in Eq. 1 is equivalent to the following condition:

$$(S_1 \models L \land S_2 \not\models L) \lor (S_1 \not\models L \land S_2 \models L)$$

(2)

where $S_1 = F(i)G(i,j)$, $S_2 = F(j)G(i,j)$, and

- $S \models L = \forall i, j : 1 \leq j < i \Rightarrow S \in L$
- $S \not\models L = \forall i, j : 1 \leq j < i \Rightarrow S \not\in L$

Note that the $S \models L$ and $S \not\models L$ are the positive and negative symbolic membership tests respectively. This version distributes the universal quantification over $i, j$ over the disjunction (proof in the appendix). Note that $S \models L$ and $S \not\models L$ are positive and negative symbolic membership tests respectively.

### 4.2 Verification using the Symbolic Membership Test

For non-regularity proof problems, the verification problem statement is: given a PDL expression $\varphi$, an expression in $E_S$ representing a congruence function $F$, and an expression in $E_S$ representing a witness function $G$, verify that $F$ and $G$ indeed are a Myhill-Nerode proof of non-regularity of $\varphi$, that is, verify that Equation 3 holds for $F$ and $G$.

We implemented symbolic membership tests for the two PDLs as automated proof systems, which are sound, but incomplete. These were precise enough to reason about all of our benchmarks.

*Symbolic Membership Oracle for UFPDL.* For a UFPDL formula $\phi$ and a symbolic string $S$, symbolic membership testing has three major steps: (1) eliminate existential quantifiers; (2) convert universal substring quantification into conjunctions; (3) symbolic evaluation over $S$ and validity checking of resultant Presburger arithmetic sentence. The symbolic non-membership test is similar, but additionally involves negating the formula at start.

*Quantifier Elimination by inductive learning.* We first eliminate all existential quantifiers by constructing Skolem functions. If $\phi = \exists v_1, v_2, v_3, \ldots v_n : \phi_{QF}$, we learn Skolem functions $f_1, f_2, \ldots f_n$ and replace all occurrences of $v_k$ in $\phi_{QF}$ by $f_k(i,j)$ where $i$ and $j$ are the variables from the symbolic string. See Example 9 for a worked out example. We do this by instantiating $i$, $j$, and quantified variables $v_k$’s of $\phi$ with small values and building a truth table using the membership query. In this truth table, we try to learn the Skolem functions. For position variables, we learn the Skolem functions as linear functions over the input domain, and for our examples, linear functions were sufficient for all
the necessary Skolem functions. For existentially quantified substring variables, we use a learning procedure from Algorithm 3 to learn symbolic strings over \( i \) and \( j \). The algorithm is explained separately in the Section 4.3. Note that in our benchmarks for non-regularity checking, no quantification over substrings was necessary.

**Example 9.** Consider the formula \( \phi = \exists p_1, p_2 : \Phi_{QF} = \exists p_1, p_2 : \text{atPos}(w, p_1) = a \land \text{atPos}(w, p_2) = b \land p_2 > p_1 \) and the symbolic string \( S = a^b \) for the positive symbolic membership test \( S \models \phi \). We first instantiate \( S \) with small values of \( i \), getting \( S(i = 1) = ab \), and \( S(i = 2) = aab \). Now, we can build the table ?? which assigns the values to \( p_1 \) and \( p_2 \) in each case. Compiling the values of \( p_1 \) and \( p_2 \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( w )</th>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( \Phi_{QF} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( ab )</td>
<td>1</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>( ab )</td>
<td>1</td>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>( ab )</td>
<td>2</td>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>2</td>
<td>( aab )</td>
<td>1</td>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>2</td>
<td>( aab )</td>
<td>1</td>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>( aab )</td>
<td>2</td>
<td>3</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 1: \( \Phi_{QF}(w, p_1, p_2) \)

which make \( \Phi_{QF}(w, p_1, p_2) \) true for each \( i \), we get \( (p_1, p_2) \in \{(1, 2)\} \) (call this set \( V(1) \)) for \( i = 1 \), and \( (p_1, p_2) \in \{(1, 3), (2, 3)\} \) (call this set \( V(2) \)) for \( i = 2 \).

Now, we inductively learn Skolem functions \( f_1 \) and \( f_2 \) for \( p_1 \) and \( p_2 \), respectively as (linear) functions of \( i \) such that \( (f_1(i), f_2(i)) \in V(i) \) for \( x \in \{1, 2\} \). In this case, we get two possibilities, i.e., \( (f_1(i) = 1, f_2(i) = i + 1) \) and \( (f_1(i) = i, f_2(i) = i + 1) \).

We substitute both possibilities for \( p_1 \) and \( p_2 \) and proceed: giving us two formula, i.e., \( \phi_1 = \text{atPos}(w, i) = a \land \text{atPos}(w, i + 1) = b \land i + 1 > 1 \) and \( \phi_2 = \text{atPos}(w, i) = a \land \text{atPos}(w, i + 1) = b \land i + 1 > i \). To complete the symbolic membership test, we need to show either \( \forall i : \phi_1(S) \) or \( \forall i : \phi_2(S) \).

**Substring variable elimination.** The set of all substrings of a symbolic string can be expressed a finite set of symbolic strings with some additional variables and constraints. For example, substrings \( \text{substr}(S) \) of \( S = 0^i(01)^j \) can be represented by the symbolic strings \{0\(^i\), 0\(^i\)(01)\(^k\), 0\(^i\)(01)\(^k\)0, 0\(^i\)(01)\(^j\), 0\(^i\)(01)\(^k\), 0\(^i\)(01)\(^k\)0, 0\(^i\)(01)\(^j\), 0\(^i\)(01)\(^k\), 0\(^i\)(01)\(^k\)0\} along with the additional constraint \( \psi = l < i \land k < j \). Therefore, universal quantification substring variables can be converted into disjunctions, i.e., \( S \models \forall w_1 : \Phi \) is equivalent to \( S \models \bigwedge_{S' \in \text{substr}(S)} \psi \land \Phi[w_1/S'] \).

**Symbolic evaluation.** Substituting the Skolem functions and eliminating and substring variables, we are left with a formula universally quantified over position
variables. We now perform symbolic evaluation of all the terms of the quantifier free $\phi_{QF}$. This is possible for each of the functions in our UFPDL (See Example 10 for a symbolic evaluation of $\text{atPos}$). The resulting expression is a formula from Presburger arithmetic whose validity is decidable.

**Example 10.** Consider $\phi = \text{isEven}(\text{length}(w)) \land \forall p \exists q. q = \text{length}(w) + 1 - p \implies \text{atPos}(w, p) \neq \text{atPos}(w, q)$ from the Benchmark 6 (see the appendix). Let $S_1 = 0^i1^j$ and $S_2 = 0^i1^j$ and we check $S_1 \models \phi$ and $S_2 \not\models \phi$. Now, $S_1 \models \phi$

\[
\begin{align*}
\equiv \forall i, p, q : q &= 2i + 1 - p \implies (\text{atPos}(S_1, p) \neq \text{atPos}(S_1, q)) \\
\equiv \forall i, p : (\text{atPos}(0^i1^j, p) \neq \text{atPos}(0^i1^j, 2i + 1 - p)) \\
\equiv \forall i, p : (p \leq i \land 2i + 1 - p > i) \lor (p > i \land 2i + 1 - p \leq i) \\
\end{align*}
\]

which is $\equiv \top$. The symbolic evaluation of $\text{atPos}$ happens as follows: for example, any fragment $\psi$ containing $\text{atPos}(S_1, p)$ is $\langle \psi[\text{atPos}(S_1, p)/0] \land p \leq i \rangle \land \langle \psi[\text{atPos}(S_1, p)/1] \land p > i \rangle$.

For $S_2 \not\models \phi$, we first build the truth table by instantiating $\phi_{QF}$ with small $i, j, p$ and $q$. From the table’s negative entries, we learn the Skolem functions for $p$ and $q$ (for our implementation, we get $p = j + 1$ and $q = i$). Using this, $0^i1^j \not\models \phi$

\[
\begin{align*}
\equiv \forall i, j, k : \neg(\text{true} \implies (\text{atPos}(0^i1^j, j + 1) \neq \text{atPos}(0^i1^j, i))) \\
\equiv \forall i, j, q : \neg(\text{true} \implies (\text{atPos}(0^i1^j, j + 1) \neq \text{atPos}(0^i1^j, i))) \equiv \top \\
\end{align*}
\]

which completes the proof of non-regularity. \qed

**Symbolic Membership Oracle for CFPDL.** We present a set of sound inference rules for the positive membership test and the negative membership test respectively for CFPDL. We write $N \rightsquigarrow S$ and $(N \to \sigma) \rightsquigarrow S$ to express that for every string in $S$ there exists a rewriting starting from the non-terminal $N$ and the rule $N \to \sigma$, respectively. Similarly, $N \not\rightsquigarrow S$ and $(N \to \sigma) \not\rightsquigarrow S$ express the non-existence of such derivations for every string in $S$. The inference rules are presented in the appendix, here we give a brief overview.

We explain the basic intuition behind the negative symbolic membership test. The rules for the positive case are simpler.

- **Top rule.** If $(N \to \sigma) \not\rightsquigarrow S$ for all occurring $\sigma$, then $N \not\rightsquigarrow S$.
- **Recurs rules.** If for all symbolic strings $(S_1, \ldots, S_k)$ with $S = S_1S_2\ldots S_k$, \forall i : N_i \not\rightsquigarrow S_i$, then $(N \to N_1\ldots N_k) \not\rightsquigarrow S$.
- **Simple rule.** If the corresponding symbols of strings $d$ and $c_1$, or $d'$ and $c_n$ do not match, $(N \to d \ldots d') \not\rightsquigarrow c_1^e \ldots c_n^e$ holds.
- **LessSymbols rule.** If no word in $\mathcal{L}(N)$ contains exactly the symbols appearing in the symbolic string $S$, $N \not\rightsquigarrow S$ holds.
- **Induction rule.** The **Induction rule** performs induction on two variables ($i$ and $j$). Let $S = c_1^e \ldots c_n^e$ with $c_1 < e_n$ and let $N \to c_1Nc_n$ be the rule in consideration. If (a) for every $0 \leq k < e_1$, $(N \to \sigma) \not\rightsquigarrow c_1^{e_1-k} \ldots c_n^{e_n-k}$ for all other rules $N \to \sigma$; and (b) $N \not\rightsquigarrow c_2^{e_2} \ldots c_n^{e_n-e_1}$, then we have $N \not\rightsquigarrow S$. 

Example 11. Consider the CFG $G \rightarrow G \mid aGa \mid bGb \mid a \mid b$ and $S_2 = b^iab^j$. We show $S_2 \not\in G$ by sketching the proof of $G \not\Rightarrow S_2$.

\[
\begin{align*}
\frac{a \neq b}{(G \rightarrow aGa|a|bGb) \not\Rightarrow ab^{i-j}} \quad \text{Simple} \\
\frac{T_1}{G \not\Rightarrow ab^{i-j}} \quad \text{Top} \\
\frac{T_1}{(G \rightarrow bGb) \not\Rightarrow b^lab^m} \quad \text{Induction} \\
\frac{G \not\Rightarrow b^lab^m}{(G \rightarrow aGa|a|bGb) \not\Rightarrow b^lab^m} \quad \text{Simple}
\end{align*}
\]

where $T_1 = (G \rightarrow aGa|a|bGb) \not\Rightarrow b^lab^m$.

The above proof states that (a) no string of the form $b^lab^m$ is derivable using the rules $G \rightarrow aGa|a|bGb$ (sub-proof $T_1$) as the symbols occurring in the rules and the string are different (rule Simple); (b) $ab^{i-j}$ is not derivable from $N$ (again, using rule Simple as the symbols in the start and end of the rules and the strings do not match); and (c) $babb$ is not derivable from $G$ and therefore, using the induction rule, neither is $b^lab^m$ for any $i > j$.

4.3 Synthesis

We present a synthesis algorithm for non-regularity proofs.

Inductive learning of symbolic strings The key part of the synthesis algorithm for non-regularity proofs involves generalizing the witness and congruence functions $F$ and $G$ as symbolic strings from specific instances. For example, given strings $F(i = 1)$ and $F(i = 2)$, to generalize it to a symbolic string $F$. To do this, we first introduce and explain a symbolic string learning algorithm which forms a key part of the learning procedure. The algorithm is presented in Algorithm 3 and is similar to the one used in [3] for learning transducers. The algorithm is explained below.

A symbolic string DAG $\langle S, s_0, s_f, E \subseteq S \times S, W \rangle$ is a directed acyclic graph with nodes $S$, edges $E$, initial and final nodes $s_0$ and $s_f$, and a labelling function $W$ which labels edges from $E$ with sets of symbolic strings. A symbolic string $S$ is contained in the DAG if and only if there is a path with edges $\delta_0, \delta_1, \ldots, \delta_n$ from $s_0$ to $s_f$ and $S = S_0S_1\ldots S_n$ such that $S_i \in W(\delta_i)$ for all $0 \leq i \leq n$. The Functions procedure takes a DAG and returns the set of symbolic strings represented by it.

Let us fix a set of variables $V$. The Generate procedure takes a string $w$ and a valuation of variables $\hat{k} \in (V \rightarrow \mathbb{N})$ and produces a compact DAG representation of all symbolic strings that produce $w$ when instantiated with the valuation $\hat{k}$. The DAG contains nodes $\{0, 1, \ldots, |w|\}$, and any symbolic string $S$ in the label of the edge from node $x$ to node $y$ has the following property that $S(\hat{k}) = \operatorname{substr}(w, x, y)$. It does so by finding all strings $u$ such that $\operatorname{substr}(w, x, y) = u^n$ for some integer $n$, and finding all functions $f$ over the variables of $\hat{k}$ such that for all $v \in V$, $f(\hat{k}(v)) = n$. For example, for a string $aa$ and $\hat{k} = \{i \mapsto 2\}$, we the set of generated edge labels are $(aa)^1$ and $a^i$ because $aa$ can be written as
either \((aa)^1\) or \(a^2\), but as \(\tilde{k}(i) = 2\), we also have \(a^i\). The functions obtained are thus \(f(i) = 1\) and \(f(i) = i\). The symbolic string DAG is in the middle part of Figure 5.

The **Intersect** procedure takes two symbolic string DAGs and intersects them so that the only symbolic strings remaining are the ones which are common. It does so by taking the product of the DAGs and labelling the edges with the intersection of the labels of the corresponding edges in each DAG.

The learning procedure **LearnSymStr** takes two tuples consisting of strings and the corresponding variable valuations \((w_j, \tilde{k}_j)\) and produces the set of symbolic strings such that for both tuples, and for all \(k_j\), \(S(k_j) = w_j\). It does so by generating the symbolic string DAGs for both tuples and intersecting them.

**Example 12.** For learning a symbolic string \(S\) over the variable \(i\) such that \(S(i = 1) = a\) and \(S(i = 2) = aa\), the working on the algorithm is in Figure 5. The only symbolic string returned is \(a^i\).

\[
\begin{align*}
\text{Fig. 5: Learning symbolic string } a^i
\end{align*}
\]

**Proof Synthesis Algorithm** We first introduce some useful notation connected with the right congruence relation \(\equiv_L\): (i) We call any word \(w \in \Sigma^*\) minimal if for all distinct prefixes \(w_1\) and \(w_2\) of \(w\), we have \(w_1 \neq_L w_2\). (ii) For any equivalence class \(n\), \(\text{Pre}(n)\) is the set of all minimal strings in \(n\). (iii) For any two different equivalence classes \(n_1\) and \(n_2\), \(\text{Post}(n_1, n_2)\) is the set of all minimal strings \(w\) such that \(w_1 w \in L\) iff \(w_2 w \notin L\) for any \(w_1 \in n_1\) and \(w_2 \in n_2\). Note that \(\text{Pre}(n)\) and \(\text{Post}(n_1, n_2)\) are always finite if \(\equiv_L\) is of finite index.

Our proof generation algorithm uses inductive synthesis to generate candidates for the congruence and witness functions. Each such candidate is tested and then verified using the membership and symbolic membership oracles.

Algorithm 3 describes an algorithm for generating a congruence function and a corresponding witness function that constitutes a proof of non-regularity of a given non-regular language \(L\). The algorithm works in the following major phases:

1. **Approximation.** (Line 2) First, we generate a finite approximation of the right congruence relation \(\equiv\) by running the \(L^*\) algorithm using counterexamples of size at most \(\text{cex}_\text{length}\).
Algorithm 3 Proving non-regularity of a language $L$

Input: PDL Expression $E$

Output: Congruence function $F$ and witness function $G$

1: $cex\_length \leftarrow 1$;
2: Let $\mathcal{A}'$ be the DFA obtained by approximation of $E$ using $L^*$ algorithm on counterexamples of size at most $k$.
3: for all equivalences $n_1, n_2$ of $\equiv_L$ and $w_1 \in n_1 \land w_2 \in n_2$:
4: for all $F \in LearnSymStr(\langle w_1, \{i \to 1\} \rangle, \langle w_2, \{i \to 2\} \rangle)$
5: $n_3 \leftarrow [F(3)]$ // where $[.]$ is the $\equiv_{L'}$ equivalence class
6: for all $w_{3,2} \in Post(n_3, n_2), w_{2,1} \in Post(n_2, n_1)$:
7: for all $G \in LearnSymStr(\langle w_{3,2}, \{i \to 3, j \to 2\} \rangle, \langle w_{2,1}, \{i \to 2, j \to 1\} \rangle)$:

8: for all $2 \leq j < i \leq 10$
9: if $(F(i)G(i, j) \in L \iff F(j)G(i, j) \in L)$
10: continue 6
11: if Verify$(F, G) \Rightarrow (F, G)$
12: else $cex\_length \leftarrow cex\_length + 1$; goto 2

LearnSymStr$(\langle s_1, \tilde{k}_1 \rangle, \langle s_2, \tilde{k}_2 \rangle) =$ Functions(Intersect$(\text{Generate}(w_1, \tilde{k}_1), \text{Generate}(w_2, \tilde{k}_2)))$

\textbf{Generate}(s, \tilde{k}) = \text{DAG}(S, s, s_t, E, W)$, where

Nodes $S = \{0, 1, \ldots, \text{Length}(s)\}$

Start/End $(s, s_t) = (0, \text{Length}(s))$

Edges $E = \{(a, b) \mid 0 \leq a < b \leq \text{Length}(s)\}$

Edge labels $W(a, b) = \{\text{substr}(s, a, b - 1)\} \cup \{w^v \mid w^k = \text{substr}(s, a, b - 1) \land (v \to k) \in \tilde{k}\}$

\textbf{Intersect}(\text{DAG}(S, s, s_t, E, W), \text{DAG}(S', s', s'_t, E', W')) = \text{DAG}(S \times S', (s, s'), (s_t, s'_t), E'', W'')$ where

$E'' = \{(s_1, s_1'), (s_2, s_2') \mid (s_1, s_2) \in E, (s_1', s_2') \in E'\}$

$W''(s, s') = W(s_1, s_2) \cap W(s'_1, s'_2)$

\textbf{Functions}(\text{DAG}(S, s, s_t, E, W)) = \{S_1 \cdot S_2 \ldots \cdot S_n \mid \delta_1, \ldots, \delta_n \text{ is a path from } s_t \text{ to } S_i, S_i \in W(\delta_i)\}$
2. **Generation.** (Lines 3–7) We use inductive learning to find candidates for the congruence and witness function. For each pair of equivalence classes $C_1$ and $C_2$ of the approximate relation, we learn possible symbolic strings $F$ from $E_S$ such that $F(i \rightarrow x) \in C_x$ for $x \in \{1, 2\}$ using the techniques of Section 4.3. For each congruence function $F$ obtained, we synthesize candidate functions for $G$ similarly with the strings $\text{Post}(F(i = 1), F(i = 2))$, $\text{Post}(F(i = 2), F(i = 3))$.

3. **Validation.** (Lines 8–10) We test each such pair of congruence and witness functions by instantiating them with small arguments and ensuring that Eq. 3 holds.

4. **Verification.** (Line 11) The validated pairs of functions are verified using the symbolic membership oracle. These are a basis of a Myhill-Nerode non-regularity proof.

5. **Refinement.** (Line 12) If the synthesis fails, we iterate with a finer approximation of relation $\equiv$ by increasing $\text{cex\_length}$.

The following theorem states that the algorithm is sound, and that it terminates if there are witness and congruence functions expressible in $E_S$. The proof can be found in the appendix.

**Theorem 6.** If $\text{SynthesizeProof}$ returns a congruence and witness function pair, they form a valid non-regularity proof. The $\text{SynthesizeProof}$ terminates, if there exist congruence and witness functions for the language $L$ in the language $E_S$.

**Example 13.** Figure 6 shows a part of the automaton obtained while learning $L = \{a^n b^n \mid n > 0\}$ using $L^\ast$. Choosing the states 1 and 2 to be the equivalence classes, we get $\text{Pre}(1) = a$ and $\text{Pre}(2) = aa$. From this we learn $F(i) = a^i$. We also get $F(3) = a^3$ and the corresponding state 3. Picking $\text{Post}(1, 2) = bb$ and $\text{Post}(2, 3) = bbb$, we learn the congruence function $G(i, j) = b^j$.

![Fig. 6: Learning $a^n b^n$](image_url)
**Characteristics.** In practice, all congruence and witness functions required in the educational setting are rather small. One can replace our complex generation procedure, and instead test/verify all pairs of small functions from $E_S$. However, testing/verification are expensive operations for complex PDLs. For example, quantified formulae in UFPDL take $0.2-0.5$ seconds to test/verify in the median case. In our experiments, the generate procedure reduced the number of function-pairs verified by over 2 orders of magnitude.

### 4.4 Experimental Evaluation

Our experiments are designed to test (a) expressivity of our PDLs, (b) expressivity of $E_S$ to produce congruence and witness functions, and (c) the power and efficiency of our proof synthesis procedure. Partial experimental data is in Table 2. Overall, we synthesized proofs for 21 problems, the rest of the data is in the appendix). The results in Table 2 are representative in terms of the time reported, the size and complexity of the witness and congruence functions generated, and other aspects.

**Expressivity of PDLs.** Out of 21 problems, only 5 were not expressible in either UFPDL or CFPDL and we consider the remaining 16 problems. Among these, 11 were directly convertible into the UFPDL, and a further 4 problems were expressible in the UFPDL after the *reverse* and *palindrome* functions were rewritten in terms of UFPDL constructs. Furthermore, 1 problem (the Dyck language) was not naturally expressible in UFPDL, but is easily expressible in CFPDL. Out of the 16 problems, 11 problems were expressible in CFPDL and 8 naturally so. More importantly, CFPDL formed a perfect complement to UFPDL in having natural formulations of all problems not directly convertible to UFPDL. In the experiments, we used a PDL for a particular problem only if there was a natural and direct formulation of the problem in it.

**Congruence and Witness Generation.** We implemented Algorithm 3. To generate the equivalence classes required, we used $L^*$ with a bound $N_\equiv$ on the number of states, rather than the length of counterexamples. We initialized $N_\equiv$ to 5 and incremented it when necessary. In most cases, the bound $N_\equiv = 5$ was sufficient, whereas for 4 problems $N_\equiv$ had to be incremented to a maximum of 10. For all our experiments, $E_S$ was expressive enough to produce valid congruence and witness functions. The functions were tested by bounding $0 < j < i < 10$ in Eq 3 and this was extremely efficient and eliminated all spurious functions.

**Congruence and Witness Verification.** The functions which pass the testing are verified using the proof systems from Section 4.1. For UFPDL, the learning procedure produced the required Skolem functions in each case and all the valid functions were verified.
5 Applications

In this section, we describe how the verification and synthesis technologies described above can assist with functionalities of ITSs.

5.1 Problem Generation

We describe how synthesis technology can assist with generating new problems. We propose a template based approach for problem generation. A template is a PDL formula with holes ⊙ instead of some operators or operands. Each hole takes values from a finite domain of operators/operands of same arity and type signature. Templates can be either provided by the teacher, or can be generated automatically by generalizing a given seed problem. An example of a set of operands of the same type that can be interchanged are \{isPrefix, isSuffix, contains\}, or the boolean connectives. For operands, string constants can be replaced by string constants of the same size or smaller. See the appendix for the full description.

The generation of concrete problems from a template by instantiating holes with values from corresponding domains requires addressing two challenges: (a) Some problems might not be well-defined (i.e., an automata construction problem might be associated with a non-regular language) or might be trivial. It is important to filter them out. (b) Due to the large number of possible instantiation it is also important to classify the non-trivial problems into different equivalence classes that are indicative of similarity between those problems. Such a partitioning can play a significant role in the workflow around an ITS. For example, if a student fails to solve a certain problem $P$, the ITS may recommend more practice problems from the same equivalence class to which $P$ belongs (after showing the reference solution for $P$). On the other hand, if a student solves a certain problem correctly, then the ITS may recommend the next challenge problem from a different equivalence class to which $P$ belongs.

We propose defining the notions of triviality of a problem and a similarity equivalence relation on problems at the level of automata for automata construction problems. Identification of trivial problems and partitioning of non-trivial problems into equivalence classes can then be easily accomplished by using the
synthesis technology. It could generate corresponding automata and abstract them using the appropriate features. Ill-defined problems can be identified by the failure to synthesize an automaton.

The notions of triviality for automata and for the similarity relations between automata are parameters that can be set by the teacher administering the tutoring system. For example, a problem can be deemed trivial if its solution is a single-state automaton (this notion is used in our experiments). The similarity equivalence relation can be any distance function between automata. In our experiments, two automata are deemed similar, if they have the same graph structure without edge labels and without considering whether a state is initial or final.

**Example 14.** Consider the problems in Example 1 and Example 2. Both of them are instances of the following generalized template:

\[
\text{count}(w, "\odot_1") \odot_2 \text{count}(w, "\odot_3")
\]

where \(\odot_1, \odot_3\) are non-empty strings, \(\odot_2 \in \{=, <, >, \leq, \geq\}\).

Most instantiations of the above template lead to non well-defined problems. Some interesting well-defined instances are: when \(\odot_3\) is a reverse string of \(\odot_1\), or \(\odot_2 \in \{=, \leq, \neq\}\) and \(\odot_1\) is a substring of \(\odot_3\) such that \(\odot_3\) contains only one instance of \(\odot_1\). Examples of trivial problems are: the case of \(\odot_1 = \odot_3\), or when \(\odot_1\) is a substring of \(\odot_3\) and \(\odot_2 \in \{\geq, <\}\).

**Experimental Evaluation.** We have implemented the problem generation and run it using the benchmark problems as seed problems. We discuss in the appendix various interesting results: number of possible instantiations of the generalized template, number of well-defined and non-trivial problems, number of equivalence classes according to the automaton structure. Due to the exponential nature of the problem generation we excluded the problems that give a very large number of instantiations. This reduction gave us 52 seed problems. Furthermore, many different problems already are instances of the same template (note that very different problems can be instances of the same template, as long as the type of the UFPDL operators and their operands match). In those cases, we keep only one template. In the end we ran the experiments with 23 different seed problems. In the median case, given a seed problem, we generate 400 problems. Out of those we generate 360 automata. Finally, we obtain 304 non-trivial problems partitioned into 12 equivalence classes. These results show that each from each seed problem it is possible to produce a number of interesting problems; thus producing a large number of practice problems.

### 5.2 Solution Checking, Generation, and Fixing

The verification technology can be used to check the correctness of a given solution. It is especially valuable in assisting with grading when the solutions to a given problem are not unique as is the case with non-regularity proofs. The synthesis technology can be used to produce reference solution(s) for a given problem. It is clearly indispensable when problems are generated automatically.
The synthesis technology can also help with suggesting fixes to an incorrect solution. For automata construction problems, we conjecture that we can seed/start the L* algorithm with an observation table extracted from the incorrect solution submitted by the student. This yields to a solution that refines the incorrect one. In case of non-regularity proof problems, if the student submits a correct congruence function but an incorrect witness function, our algorithm can generate an appropriate witness function. Detailed investigation of utility of these fixing methodologies is left for future work.

6 Related Work

Gulwani et al. [4] presented a synthesis technology for generating solutions to high-school geometry construction problems. The work presents an inductive learning algorithm that involves generating few models and then generalizes models into a construction. Although, we use inductive learning here, the problem domains and the synthesis algorithms are completely different. Moreover, in [4], verification of synthesized results is not possible (in contrast to our case, where our synthesis technology builds over the verification technology). Gulwani [3] presented an inductive learning algorithm based on version-space algebras for learning syntactic string transformations. Our algorithm for generating witness functions (in case of non-regularity proof generation) uses a DAG based data-structure (similarly to [3]). However, our underlying language of witness functions (which map integers to strings) is different from the underlying language in case of string to string transformations. Template based problem generation has mostly been explored in the context of algebra problems. In [6], holes range over constants and a random instantiation generates a valid problem. In [14], holes range over both algebraic operators and constants in a proof problem, and random testing is used to filter out invalid problems. We not only take the idea of template based problem generation to a new domain of automata theory problems, but also present an approach for classifying the problems into different equivalence classes based on a parameterized notion of similarity.

We note that a large part of UFPDL can be captured in previously studied logics, such as Monadic Second Order (MSO) with Cardinalities [8]. However, we opt for the UFPDL to enable a more direct translation from natural language. We do not know of any results of whether the problem of regularity for MSO with cardinalities is decidable. The fragments for which decision procedures exist include deterministic pushdown automata [15].

JFLAP [12] is a mature system used widely for teaching automata theory and formal language theory. In contrast to our notion of PDLs, JFLAP accepts standard formal descriptions as input such as regular expressions and automata. Our focus is on core conceptual content of automata construction and non-regularity proof construction, and on synthesis of solution automata. We address the problem generation aspect of ITS that is not addressed by JFLAP.

Finally, there is a rich literature on ITSs in education and artificial intelligence communities. We refer the reader to survey [10] and textbooks [16] for an
overview of the field. However, our work is perhaps the only one besides JFLAP to focus on the domain of automata theory. Our problem solving and problem generation techniques both leverage domain-specific insights.

7 Conclusion and Future Work

The need for intelligent tutoring systems (ITS) to extend the reach of high-quality education by making it more interactive and customized is indisputable. Here, we have provided the building blocks of an ITS for two classic problems in automata theory course.

There are several directions of future work. First, we are working on integrating a natural language processing front-end to convert English problem descriptions to UFPDL. Second, we plan to use field data to increase the potential impact of our techniques. We started engaging with some professors teaching finite automata theory to deploy our system in their classrooms as part of various workflows. The data collected can help us measure the effectiveness of various workflows of the tool. Furthermore, once we have statistical data about perceived problem difficulty, we can use machine learning techniques for recommending problems. We plan to work on a better feedback system that also points out conceptual mistakes (such as “incorrect behavior on empty string or small strings”, “improper handling of error state”), based on real student data. Third, we want to extend the verification and synthesis technologies for automata construction described in this paper to other useful concepts such as regular expressions and CFGs.

References

The \( L^* \) algorithm learns an unknown regular language and generates a minimal DFA that accepts the regular language. This algorithm was introduced by Angluin [1], but we use an improved version by Rivest and Schapire [11].

The algorithm infers the structure of the DFA by asking a teacher, who knows the unknown language, two types of questions: membership queries and equivalence queries. On a membership query, the learner asks whether a string \( \sigma \) is accepted by the unknown language, and the teacher answers \( \text{true} \) or \( \text{false} \). On an equivalence query, the learner conjectures that the machine it has constructed is equivalent to the unknown language. The teacher replies that the conjecture is either correct or incorrect, and in the latter case gives a counter-example which is a string accepted by one but not the other.

Figure 7 illustrates the \( L^* \) algorithm. Let \( U \) be the unknown regular language and \( \Sigma \) be its alphabet. At any given time, the \( L^* \) algorithm has information about a finite collection of strings over \( \Sigma \), classified either as members or non-members of \( U \). This information is maintained in an observation table \((S, E, T)\) where \( S \) and \( E \) are a set of strings over \( \Sigma \), and \( T \) is a function from \((S \cup S \cdot \Sigma) \cdot E\) to \{\text{true}, \text{false}\}. Intuitively, \( S \) can be viewed as a set of representative strings that lead from the initial state (uniquely) to the various states of the DFA, and \( E \) as experiments that are performed at these states in order to distinguish states. \( T \) maps strings \( \sigma \) in \((S \cup S \cdot \Sigma) \cdot E\) to \text{true} if \( \sigma \) is in \( U \), and to \text{false} otherwise. Initially, \( S \) and \( E \) are set to \{\( \varepsilon \)\}, and \( T \), which is implemented as a two-dimensional array, is initialized using membership queries for every string in \((S \cup S \cdot \Sigma) \cdot E\) (line 2–5). In line 7, it checks whether the observation table is closed; that is, for every \( s \in S \) and \( a \in \Sigma \), there exists \( s' \in S \) such that \( T[s \cdot a, e] = T[s', e] \) for every \( e \in E \). If not, each such \( s \cdot a \) (e.g., \( s_{\text{new}} \) in line 8) is simply added to \( S \).

The algorithm again updates \( T \) with regard to \( s \cdot a \) (line 9–11). Once the table is closed, it constructs a conjecture machine \( C = (Q, q_0, F, \delta) \) as follows (line 13):
\textbf{L* Algorithm}

1: \( S := \{ \epsilon \}; \ E := \{ \epsilon \}; \)
2: \textbf{foreach} \((s \in S), (a \in \Sigma) \) and \((e \in E) \) \{ 
3: \quad \( T[s,e] := \text{Member}(s \cdot e); \)
4: \quad \( T[s \cdot a, e] := \text{Member}(s \cdot a \cdot e); \)
5: \} 
6: \textbf{repeat:}
7: \quad \textbf{while} \((s_{\text{new}} := \text{Closed}(S, E, T)) \neq \text{null}) \{ 
8: \quad \quad \text{Add}(S, s_{\text{new}}); 
9: \quad \quad \textbf{foreach} \((a \in \Sigma) \) and \((e \in E) \) \{ 
10: \quad \quad \quad \( T[s_{\text{new}} \cdot a, e] := \text{Member}(s_{\text{new}} \cdot a \cdot e); \)
11: \quad \quad \} 
12: \quad \} 
13: \quad \( C := \text{MakeConjectureMachine}(S, E, T); \)
14: \quad \textbf{if} \((\text{cex} := \text{Equivalent}(C)) = \text{null}) \textbf{then return} C; 
15: \quad \textbf{else} \{ 
16: \quad \quad e_{\text{new}} := \text{FindSuffix}(\text{cex}); 
17: \quad \quad \text{Add}(E, e_{\text{new}}); 
18: \quad \quad \textbf{foreach} \((s \in S) \) and \((a \in \Sigma) \) \{ 
19: \quad \quad \quad \( T[s, e_{\text{new}}] := \text{Member}(s \cdot e_{\text{new}}); \)
20: \quad \quad \quad \( T[s \cdot a, e_{\text{new}}] := \text{Member}(s \cdot a \cdot e_{\text{new}}); \)
21: \quad \quad \} 
22: \quad \} 

Fig. 7: L* algorithm

\( Q = S, q_0 = \epsilon, \ F = \{ s \in S \mid T[s, \epsilon] = \text{true} \}, \) and for every \( s \in S \) and \( a \in \Sigma, \) \( \delta(s, a) = s' \) such that \( T[s \cdot a, e] = T[s', e] \) for every \( e \in E. \) Finally, if the answer of the equivalence query is yes, it returns the current machine \( C; \) otherwise, a counter-example \( \text{cex} \in ((L(C) \setminus U) \cup (U \setminus L(C))) \) is provided by the teacher. The algorithm analyzes the counter-example \( \text{cex} \) in order to find the longest suffix \( e_{\text{new}} \) of \( \text{cex} \) that witnesses a difference between \( U \) and \( L(C) \) (line 16). Adding \( e_{\text{new}} \) to \( E \) reflects the difference in the next conjecture by splitting a state in \( C. \) It then updates \( T \) with respect to \( e_{\text{new}}. \)

The \( L^* \) algorithm guarantees to construct a minimal DFA for the unknown regular language using only a polynomial number of membership and equivalence queries: more precisely with \( O(|\Sigma|^n^2 + n \log m \) membership queries and at most \( n - 1 \) equivalence queries, where \( n \) is the number of states in the final DFA and \( m \) is the length of the longest counter-example provided by the teacher for equivalence queries.
B Proofs for Results in Section 3

The following result (Proposition 1) allows us to bound the size of a counterexample to an equivalence check between a formula in a PDL and an automaton. To prove it, we first need to prove another proposition.

**Proposition 2.** Let $L \subseteq \Sigma^*$ be a regular language. Let $A = (Q, \Sigma, q_0, F)$ be a minimal deterministic automaton such that $L(A) = L$, and let $k$ be the number of states of $A$. Let $q_1$ and $q_2$ be two distinct states of $A$. There exists a string $w \in \Sigma^*$ such that $|w| \leq k - 2$ and $w \in L_A(q_1)$ iff $w \not\in L_A(q_2)$.

*Proof.* The proposition is proven by analysis of the standard minimization algorithm (see [5]).

Let $\phi$ be a functional on relations over $Q$ defined by $\varphi(R) = \{((q_1, q_2) \mid (q_1 \in F_S \text{ iff } q_2 \in F_S) \lor ((q_1, q_2) \in R) \land \forall a \in \Sigma_S (\delta_S(q_1, a), \delta_S(q_2, a)) \in R\}$. Let us consider the following family of relations: $R_0 = \{((q_1, q_2) \mid q_1 \in F_S \text{ iff } q_2 \in F\}$, and $R_{i+1} = \varphi(R_i)$. For all $i$, $R_i$ is an equivalence relation.

From the definition of $\phi$, we have that for all $i$, each equivalence class of $R_{i+1}$ is a subset of an equivalence class of $R_i$. Therefore, for all $i$, if $R_{i+1} \neq R_i$, then the number of equivalence classes is strictly greater than the number of equivalence classes of $R_i$, and $R_{i+1} = \varphi(R_i)$. For all integers $j$ greater than $k - 2$, we have that $R_j = R_{k-2}$.

We now show that for all states $q_1$ and $q_2$ in $Q$, for all $i$, if $q_1$ and $q_2$ are not $R_i$-equivalent, then there exists a word $w$ of length at most $i$ such that $w \in L_A(q_1)$ iff $w \not\in L_A(q_2)$. We proceed by induction on $i$. For $i = 0$, we can take the empty word. To prove the inductive case, let us consider two states $q_1$ and $q_2$ such that $q_1$ and $q_2$ are not $R_{i+1}$-equivalent. We consider two sub-cases. First, if $q_1$ and $q_2$ are not $R_i$-equivalent, then we can use the induction hypothesis. Second, let us suppose that $q_1$ and $q_2$ are $R_i$-equivalent. This, together with the fact that $q_1$ and $q_2$ are not $R_{i+1}$-equivalent, implies that there exists $a \in \Sigma_S$ and two states $q'_1$ and $q'_2$ such that $\delta_S(q_1, a) = q'_1$, $\delta_S(q_2, a) = q'_2$, and $(q'_1, q'_2) \not\in R_i$. Let us use the induction hypothesis for $(q'_1, q'_2)$ to obtain a word $w'$ such that $w' \in L_A(q'_1)$, but $w' \not\in L_A(q'_2)$. We then construct $w$ as $aw'$. It’s length is at most $i + 1$.

It remains to show that for all $q_1$ and $q_2$, $q_1$ and $q_2$ are not are not $R_{k-2}$-equivalent. This follows from the argument use in the standard minimization construction.

**Proposition 1** Let $L \subseteq \Sigma^*$ be a regular language. Let $A$ be a minimal deterministic automaton such that $L(A) = L$, and let $k$ be the number of states of $A$. For all strings $u$ and $v$ in $\Sigma^*$, if $u \equiv_L v$ then $u \equiv_L v$.

*Proof.* Let us consider two strings $u$ and $v$. If $\delta^*(q_0, u) = \delta^*(q_0, v)$, then $u \equiv_L v$. Let us now assume that $\delta^*(q_0, u) = q_1$, $\delta^*(q_0, v)$, and $q_1 \neq q_2$. By Proposition 2, we have that there exists a string $w \in \Sigma^*$ such that $|w| \leq k - 2$ and $w \in$
$L_A(q_1) \ n \in L_A(q_2)$. However, this implies that it is not the case that $u \equiv_L v$, which allows us to finish the proof.

**Theorem 5.** [Alternate Form of Myhill-Nerode Theorem] The condition in Eq. 1 is equivalent to the following condition:

$$ (S_1 \models L \land S_2 \not\models L) \lor (S_1 \not\models L \land S_2 \models L) \quad (3) $$

where $S_1 = F(i) G(i, j)$, $S_2 = F(j) G(i, j)$, $S \models L \triangleq \forall i, j : 1 \leq j < i \Rightarrow S \in L$, $S \not\models L \triangleq \forall i, j : 1 \leq j < i \Rightarrow S \not\in L$.

Note that the $S \models L$ and $S \not\models L$ are the positive and negative symbolic membership tests respectively.

**Proof.** It is easy to show that Eq. 3 implies Eq. 1. We now prove the other direction. Suppose $L$ is non-regular. Let $F^*$ and $G^*$ be the functions satisfying Eq. 1. Define $\text{Comp}(w) = \{x|wx \in L\}$ and let $\text{Comps} = \{(n, \text{Comp}(w))|n \in \mathbb{N} \land w = F^*(n)\}$. Consider the partial order $\langle \text{Comps}, \leq \rangle$ where $(n, S_n) \leq (m, S_m) \iff S_n \subseteq S_m$.

First we can easily show that $\langle \text{Comps}, \leq \rangle$ contains an infinite strictly ascending or descending chain or an infinite anti-chain. As $F^*$ satisfy Eq. 1, $\text{Comps}$ is an infinite set. By Ramsey’s theorem for pairs we know that every countable partial-order (and hence, $\langle \text{Comps}, \leq \rangle$) has either an infinite chain or an infinite anti-chain. We can now check what happens in either case:

1. If $\langle \text{Comps}, \leq \rangle$ contains an infinite strictly ascending chain, i.e., $(n_1, S_{n_1}) < (n_2, S_{n_2}) < (n_3, S_{n_3}) \ldots$, we let $F(i) = F^*(n_i)$ and pick $G(i, j)$ as any string from the non-empty set $S_{n_i} \setminus S_{n_j}$.
2. If $\langle \text{Comps}, \leq \rangle$ contains an infinite strictly descending chain, i.e., $(n_1, S_{n_1}) > (n_2, S_{n_2}) > (n_3, S_{n_3}) \ldots$, we let $F(i) = F^*(n_i)$ and pick $G(i, j)$ as any string from the non-empty set $S_{n_j} \setminus S_{n_i}$.
3. If $\langle \text{Comps}, \leq \rangle$ does not contain any infinite chain, it contains an infinite anti-chain. Let this infinite anti-chain be $\{(n_k, S_{n_k})|k \in \mathbb{N}\}$. We let $F(i) = F^*(n_i)$ and pick $G(i, j)$ as any string from the non-empty set $S_{n_j} \setminus S_{n_i}$.

In each case, it is easy to show that Eq. 3 holds for the defined $F$ and $G$.

C Additional Details for Section 4.2

The notation $(N \rightarrow \sigma) \leadsto S$ and $N \rightarrow S$ denote that there exists a derivation of the symbolic string by the rules of grammar $G$ starting with the rule $N \rightarrow \sigma$ and from non-terminal $N$, respectively. Without loss of generality, assume that no
\[
\begin{align*}
N \rightarrow \sigma \rightarrow S & \quad \text{Top} \\
N \rightarrow S & \\
\exists ((S_1, S_2, \ldots, S_k) : k\text{-partition of } S) \forall n < k : N_n \rightarrow S_n & \quad \text{Recur} \\
(N \rightarrow c_1^{i_1} N_{c_2}^{i_2} \ldots N_{c_n}^{i_n}) & \rightarrow c_1^{e_1} \ldots c_n^{e_n} & \quad \text{Induction}
\end{align*}
\]

\[
I_1 \quad I_2 \quad B
\]

\[
(N \rightarrow c_1^{i_1} N_{c_2}^{i_2} \ldots N_{c_n}^{i_n}) \rightarrow c_1^{e_1} \ldots c_n^{e_n}
\]

\[
\begin{align*}
I_1 & \overset{\text{def}}{=} \forall i_0, j_0 : 1 \leq j_0 < i_0 \Rightarrow \exists i, j : (1 \leq j < i \leq i_0 \land j < j_0) \\
& \quad \land \left( \bigwedge_{k=1}^{n} (e_k[(i_0 + 1)/i, j_0/j] - f_k = e_k) \right) \\
I_2 & \overset{\text{def}}{=} \forall i_0, j_0 : 1 \leq j_0 < i_0 - 1 \Rightarrow \exists i, j : (1 \leq j < i \leq i_0 \land j \leq j_0) \\
& \quad \land \left( \bigwedge_{k=1}^{n} (e_k[i_0/i, (j_0 + 1)/j] - f_k = e_k) \right) \\
B & \overset{\text{def}}{=} c_1^{e_1[2/i,1/j]} \ldots c_n^{e_n[2/i,1/j]} \vdash N
\end{align*}
\]

Fig. 8: Positive Symbolic Membership Test for CFG

Half-terminal in grammar $G$ can be rewritten to the empty string. As none of our symbolic strings can represent the empty string, we can replace any grammar with such a grammar.

We explain the rules below:

- **Top rules.** It is sufficient to establish positive symbolic membership from one of the rules $N \rightarrow \sigma$ in the grammar, while the negative symbolic membership test requires establishing that for all the rules $N \rightarrow \sigma_1, \ldots, N \rightarrow \sigma_n$ there is no derivation of the symbolic string.

- **Induction rules.** Both Induction rules perform induction on two variables ($i$ and $j$). The rules are complicated by the additional constraint $i > j$. The positive rule states that if the symbolic string for the base case $i = 2 \land j = 1$ is rewritable from $N$, all strings can be obtained by recursively applying the rule $N \rightarrow c_1^{i_1} N_{c_2}^{i_2} \ldots N_{c_n}^{i_n}$ repeatedly and then rewriting to the base case. Furthermore, for the negative membership test, it requires to establish that neither the symbolic string, nor any of the inductively obtained substrings can be generated using other rules.

- **Simple rule.** This rule says that a string $c_1^{e_1} \ldots c_n^{e_n}$ cannot be generated by $N \rightarrow d \ldots d'$ if the corresponding symbols of $d$ and $c_1$, and $d'$ and $c_n$ do not match.

- **Recur rules.** Let a $k$-partition of string $w$ be $(w_1, w_2, \ldots, w_k)$ such that $w = w_1 w_2 \ldots w_k$. As for the case of substrings, the set of all partitions of a symbolic string can be expressed using symbolic strings. The positive (negative) Recur rule states that a symbolic string is derivable (not deriv-
\( (N \rightarrow \sigma_1) \not\vdash S \ldots (N \rightarrow \sigma_n) \not\vdash S \)  
\[ \text{Top} \]

\[ \forall ((S_1, S_2, \ldots, S_k) :: k\text{-partition of } S) \forall n < k : N_n \not\vdash S_n \]  
\[ (N \rightarrow N_1 N_2 \ldots N_k) \rightarrow S \]  
\[ \text{Recur} \]

\( \neg \text{isPrefix}(d, c_1) \lor \neg \text{isSuffix}(d', c_n) \)  
\[ (N \rightarrow dsd') \not\vdash c_1^{e_1} \ldots c_n^{e_n} \]  
\[ \text{Simple} \]

\[ \text{symbols}(S) \neq \text{symbols}(w) : \forall w \in L(N) \]  
\[ N \not\vdash S \]  
\[ \text{LessSymbols} \]

\( I_1 \quad I_2 \quad B \quad N \not\vdash c_1^{e_1} \ldots c_n^{e_n-1} \quad N \not\vdash c_2^{e_2} \ldots c_n^{e_n-1} \)

\[ \forall \sigma \neq (c_1^{i_0} Nc_n^{j_0}) : N \rightarrow \sigma \not\vdash c_1^{e_1} \ldots c_n^{e_n-k} \]
\[ (N \rightarrow c_1^{i_0} Nc_n^{j_0}) \not\vdash c_1^{e_1} \ldots c_n^{e_n} \]  
\[ \text{Induction} \]

\[ I_1 \overset{\text{def}}{=} \forall i_0, j_0 : 1 \leq j_0 < i_0 \Rightarrow \exists i, j : (1 \leq j < i \leq i_0 \land j \leq j_0 \land \]
\[ \bigwedge_{k=1}^{n} (e_k' > 0 \Rightarrow e_k'' = e_k)) \), where \( e_k' \equiv e_k[(i_0 + 1)/i, j_0/j] - f_k \)

\[ I_2 \overset{\text{def}}{=} \forall i_0, j_0 : 1 \leq j_0 < i_0 - 1 \Rightarrow \exists i, j : (1 \leq j < i \leq i_0 \land j \leq j_0 \land \]
\[ \bigwedge_{k=1}^{n} (e_k'' > 0 \Rightarrow e_k'' = e_k)) \), where \( e_k'' \equiv e_k[i_0/i, (j_0 + 1)/j] - f_k \)

\[ B \overset{\text{def}}{=} c_1^{e_1[2/i,1/j]} \ldots c_n^{e_n[2/i,1/j]} \not\vdash N \]

Fig. 9: Negative Symbolic Membership Test for CFG
able) through a rule \( N \rightarrow N_1N_2\ldots N_k \) if there exists (for all) \( k \)-partitions of the symbolic string, the \( i^{th} \) component of the partition is derivable (not derivable) from \( N_i \).

- **LessSymbols rule.** The **LessSymbols** relies on a simple sufficient observation to establish that a symbolic string is not derivable from a non-terminal \( N \) if all the terminals that occur in rewritings of \( N \) do not contain some terminal in the symbolic string. This is an incomplete rule to complement **Simple** and eliminates rules in which the beginnings and ends match (i.e., it cannot be eliminated by simple) but, never generates the required terminals for the symbolic string.

**Theorem.** (Soundness)

- If **SynthesizeProof** returns a congruence and witness function pair, they form a valid non-regularity proof.
- The **SynthesizeProof** terminates and returns a congruence and witness function pair, if there exist congruence and witness functions for the language \( L \) in the language \( E_S \).

**Proof.** Soundness is immediate. The completeness result from the following argument. Suppose \( F \) and \( G \) are any congruence and witness functions for \( L \) that are expressible in language \( E_S \) and can be verified by the symbolic membership oracle associated with the PDL. Then, the algorithm will construct those at line 7 for \( k = \text{Max}(\text{Length}(F(3)G(3,2)), \text{Length}(F(2)G(2,1))) \).

### D Additional Details for Section 4.4

Table 3 contains additional experimental details.

### E Detail of the Learning Experiments

#### E.1 Benchmarks Statements (regular languages).

Here we present the benchmark examples collected for the synthesis part of our experimental evaluation. The UFPLDL translation is also given when it exists. We collected benchmarks from automata theory courses taught to undergraduate students. We examined the course material designed by Prof. Rajeev Alur at University of Pennsylvania, Prof. Sanjit A. Seshia at UC Berkeley, Prof. Jeffrey D. Ullman’s at Stanford, Dr. Mary Eberlein’s at UT Austin, Dr. Elaine Rich at UT Austin, Prof. Jeff Foster at University of Maryland, Goutam Biswas at IIT Kharagpur, Prof. Tao Jiang at UC Riverside, Dr. Timo Kötzing at University of Delaware, Prof. Jean Gallier at University of Pennsylvania, Prof. Robert Sedgewick and Dr. Kevin Wayne at Princeton University. We also examined some books [5, 7]. We collected the problems that roughly follow the template “draw an automata for ...”.

The PDL used in our implementation has some minor difference compared to the UFPLDL as it is presented. There are a few additional predicates which
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(2)</td>
<td>(3)</td>
<td>(4)</td>
<td>(5)</td>
<td>(6)</td>
<td>(7)</td>
<td>(8)</td>
</tr>
<tr>
<td>1</td>
<td>UFPDL</td>
<td>5</td>
<td>0.14</td>
<td>$a^i$</td>
<td>$b^j$</td>
<td>$&lt;0.05$</td>
<td>$&lt;0.05$</td>
</tr>
<tr>
<td>2</td>
<td>UFPDL</td>
<td>5</td>
<td>0.17</td>
<td>$a^i$</td>
<td>$b^j$</td>
<td>$&lt;0.05$</td>
<td>$&lt;0.05$</td>
</tr>
<tr>
<td>3</td>
<td>UFPDL</td>
<td>5</td>
<td>0.85</td>
<td>$a^i$</td>
<td>$b^j$</td>
<td>0.13</td>
<td>$&lt;0.05$</td>
</tr>
<tr>
<td>3</td>
<td>CFPDL</td>
<td>5</td>
<td>0.38</td>
<td>$a^i$</td>
<td>$b^j$</td>
<td>$&lt;0.05$</td>
<td>$&lt;0.05$</td>
</tr>
<tr>
<td>4</td>
<td>UFPDL</td>
<td>5</td>
<td>0.16</td>
<td>$b^j$</td>
<td>$a^j$</td>
<td>0.81</td>
<td>0.15</td>
</tr>
<tr>
<td>5</td>
<td>CFPDL</td>
<td>10</td>
<td>0.18</td>
<td>$a^ib^j$</td>
<td>$ba^j$</td>
<td>$&lt;0.05$</td>
<td>$&lt;0.05$</td>
</tr>
<tr>
<td>6</td>
<td>CFPDL</td>
<td>5</td>
<td>0.11</td>
<td>$b^j$</td>
<td>$a^j$</td>
<td>$&lt;0.05$</td>
<td>$&lt;0.05$</td>
</tr>
<tr>
<td>9</td>
<td>CFPDL</td>
<td>10</td>
<td>0.06</td>
<td>$a^ib^j$</td>
<td>$a^j$</td>
<td>$&lt;0.05$</td>
<td>$&lt;0.05$</td>
</tr>
<tr>
<td>11</td>
<td>UFPDL</td>
<td>8</td>
<td>12.1</td>
<td>$ab^j$</td>
<td>$c^j$</td>
<td>0.40</td>
<td>0.11</td>
</tr>
<tr>
<td>12</td>
<td>UFPDL</td>
<td>5</td>
<td>10.1</td>
<td>$(ab)^j$</td>
<td>$a^j$</td>
<td>0.42</td>
<td>0.23</td>
</tr>
<tr>
<td>13</td>
<td>UFPDL</td>
<td>5</td>
<td>0.64</td>
<td>$a^i$</td>
<td>$b^j$</td>
<td>$c^j$</td>
<td>0.15</td>
</tr>
<tr>
<td>15</td>
<td>UFPDL</td>
<td>7</td>
<td>0.78</td>
<td>$ab^j$</td>
<td>$a^j$</td>
<td>1.11</td>
<td>0.08</td>
</tr>
<tr>
<td>15</td>
<td>CFPDL</td>
<td>7</td>
<td>0.13</td>
<td>$a^i$</td>
<td>$ba^j$</td>
<td>$&lt;0.05$</td>
<td>$&lt;0.05$</td>
</tr>
<tr>
<td>16</td>
<td>CFPDL</td>
<td>5</td>
<td>0.24</td>
<td>$a^i$</td>
<td>$b^j$</td>
<td>$&lt;0.05$</td>
<td>$&lt;0.05$</td>
</tr>
<tr>
<td>17</td>
<td>UFPDL</td>
<td>5</td>
<td>0.16</td>
<td>$a^i$</td>
<td>$b^j$</td>
<td>0.44</td>
<td>0.14</td>
</tr>
<tr>
<td>18</td>
<td>CFPDL</td>
<td>5</td>
<td>0.27</td>
<td>$a^ib^j$</td>
<td>$a^j$</td>
<td>$&lt;0.05$</td>
<td>$&lt;0.05$</td>
</tr>
<tr>
<td>19</td>
<td>UFPDL</td>
<td>10</td>
<td>15.8</td>
<td>$ab^j$</td>
<td>$c^j$</td>
<td>4.5</td>
<td>1.7</td>
</tr>
<tr>
<td>21</td>
<td>CFPDL</td>
<td>5</td>
<td>0.51</td>
<td>$(i^j)$</td>
<td>$j^j$</td>
<td>0.11</td>
<td>$&lt;0.05$</td>
</tr>
</tbody>
</table>

Table 3: Non-regularity proofs (1) ID, (2) PDL, (3) $N_{\equiv}$ required, (3) cong./witness generation (sec), (5) generated cong. function, (6) generated witness function, (7) testing (sec), (8) verification (sec).
are syntactic sugar, e.g. isEven, isOdd. A few predicates have a different name, e.g. card becomes size. Finally, count is expanded into size(indexesOf(_)). Furthermore, we explicitly refer to the underlying model as the string $w$.

The examples are grouped by provenance.

By Prof. Rajeev Alur at University of Pennsylvania

**Benchmark 1**

Statement: The set of strings $w$ such that the symbol at every odd position in $w$ is $a$.

Formula: $\forall p. (1 <= p \land p <= \text{length}(w) \land \text{isOdd}(p)) \Rightarrow \text{atPos}(w, p) = 'a'$

Alphabet: a b

**Benchmark 2**

Statement: The set of strings $w$ such that the number of occurrences of the substring $ab$ in $w$ equals the number of occurrences of the substring $ba$ in $w$.

Formula: $\text{size}(\text{indexesOf}(w, "ab")) = \text{size}(\text{indexesOf}(w, "ba"))$

Alphabet: a b

**Benchmark 3**

Statement: Consider the language $L$ consisting of strings $w$ such that $w$ has odd number of $a$ symbols and even number of $b$ symbols.

Formula: $\text{isOdd}(\text{size}(\text{indexesOf}(w, \text{\textquote{a}}))) \land \text{isEven}(\text{size}(\text{indexesOf}(w, \text{\textquote{b}})))$

Alphabet: a b

**Benchmark 4**

Statement: For natural numbers $m$ and $n$, consider the parameterized language $L(m,n)$ consisting of strings $w$ such that $w$ contains at least $m$ occurrences of the symbol $a$ and at most $n$ occurrences of the symbol $b$.

Formula: --

Alphabet: a b

**Benchmark 5**

Statement: Consider the set $L$ of strings $w$ such that the last symbol of $w$ has not appeared before.

Formula: $\neg \in(\text{fromEnd}(w, 1), \text{symbols}(\text{substring}(w, 1, \text{length}(w))))$

Alphabet: a b

**Benchmark 6**

Statement: Consider the language $L$ consisting of words that contain 010

Formula: $\text{contains}(w, \text{\textquote{010}})$

Alphabet: 0 1
Benchmark 7
Statement: consider the language L containing words that begin with a and end with b.
Formula: atPos(w, 1) = 'a' && fromEnd(w, 1) = 'b'
Alphabet: a b

Benchmark 8
Statement: Let L1 be the set of words w that contain an even number of a's. Let L2 be the set of words w that end with b. Let L3 = L1 intersect L2.
Formula: isEven(size(indexesOf(w, 'a'))) && fromEnd(w, 1) = 'b'
Alphabet: a b

Benchmark 9
Statement: Consider the language L consisting of strings w such that the set of symbols that appear in w does not equal a,b,c
Formula: symbols(w) != {'a','b','c'}
Alphabet: a b c

Benchmark 10
Statement: A string w belongs to the language L precisely when w contains at least one a symbol and does not contain any b symbols.
Formula: contains(w, "a") && !contains(w, "b")
Alphabet: a b

Benchmark 11
Statement: Consider the language L consisting of words w such that the length of w is odd if and only if w ends with the symbol a.
Formula: isOdd(length(w)) = (fromEnd(w, 1) = 'a')
Alphabet: a b

From [5]
Benchmark 12
Statement: The set of all strings ending in 00
Formula: isSuffix("00", w)
Alphabet: 0 1

Benchmark 13
Statement: The set of all strings with three consecutive 0's (not necessarily at the end)
Formula: contains(w, "000")
Alphabet: 0 1
**Benchmark 14**
Statement: The set of strings with 011 as a substring
Formula: `contains(w, "011")`
Alphabet: 0 1

**Benchmark 15**
Statement: Set of all strings such that each block of five consecutive symbols contains at least two 0's
Formula: `forall p. (1 <= p & p <= length(w) - 4) => size(indexesOf(substring(w, p, p + 5), "0")) >= 2`
Alphabet: 0 1

**Benchmark 16**
Statement: Set of all strings whose tenth symbol from the right end is a ‘1’.
Formula: `fromEnd(w, 10) = '1'`
Alphabet: 0 1

**Benchmark 17**
Statement: Set of strings that either begin or end (or both) with 01.
Formula: `isPrefix("01", w) || isSuffix("01", w)`
Alphabet: 0 1

**Benchmark 18**
Statement: Set of strings such that the number of 0’s is divisible by 5 and the number of 1’s is divisible by 3.
Formula: `size(indexesOf(w, "0")) % 5 = 0 && size(indexesOf(w, "1")) % 3 = 0`
Alphabet: 0 1

**Benchmark 19**
Statement: The set of strings over alphabet 0,1,...,9 such that the final digit has not appeared before.
Formula: `!in(fromEnd(w, 1), symbols(substring(w, 1, length(w))))`
Alphabet: 0 1 2 3 4 5 6 7 8 9

**Benchmark 20**
Statement: The set of strings of 0’s and 1’s such that there are two 0’s separated by a number of positions that is a multiple of 3.
Formula: `exists p1 p2. p1 > p2 && atPos(w, p1) = '0' && atPos(w, p2) = '0' && (p2-p1) % 3 = 0`
Alphabet: 0 1
**Benchmark 21**

Statement: The set of strings consisting of zero or more a’s followed by zero or more b’s, followed by zero or more c’s.

Formula: $match(a^w_1 b^w_2 c^w_3, w, \text{true})$

Alphabet: a b c

**Benchmark 22**

Statement: The set of strings that consist of either 01 repeated one or more times or 010 repeated one or more times.

Formula: $match((01)^x, w, x > 0) \lor match((010)^x, w, x > 0)$

Alphabet: 0 1

**Benchmark 23**

Statement: The set of strings of 0s and 1s such that at least one of the last ten positions is a 1.

Formula: $\exists x. 0 < x \&\& x <= 10 \&\& \text{fromEnd}(w, x) = '1'$

Alphabet: 0 1

**Benchmark 24**

Statement: The set of strings over alphabet a,b,c containing at least one a and at least one b.

Formula: $\text{contains}(w, 'a') \&\& \text{contains}(w, 'b')$

Alphabet: a b c

**Benchmark 25**

Statement: The set of strings of 0s and 1s whose tenth symbol form the right end is 1.

Formula: $\text{fromEnd}(w, 10) = '1'$

Alphabet: 0 1

**Benchmark 26**

Statement: The set of strings of 0s and 1s with at most one pair of consecutive 1s.

Formula: $\text{size}(\text{indexesOf}(w, '11')) <= 1$

Alphabet: 0 1

**Benchmark 27**

Statement: The set of all strings of 0s and 1s such that every pair of adjacent 0s appears before any pair of adjacent 1s.

Formula: $\forall p1 p2. (\text{in}(p1, \text{indexesOf}(w, '00')) \&\& \text{in}(p2, \text{indexesOf}(w, '11'))) \Rightarrow p1 < p2$

Alphabet: 0 1
Benchmark 28
Statement: The set of strings of 0s and 1s whose number of 0s is divisible by five.
Formula: \( \text{size(indexesOf}(w, "0")) \% 5 = 0 \)
Alphabet: 0 1

Benchmark 29
Statement: The set of all strings of 0s and 1s not containing 101 as a substring.
Formula: \!contains(w, "101")
Alphabet: 0 1

Benchmark 30
Statement: The set of all strings with an equal number of 0s and 1s such that no prefix has two more 0s than 1s, nor two more 1s than 0s.
Formula: \( \text{size(indexesOf}(w, "0")) = \text{size(indexesOf}(w, "1")) && \text{forall } p. (1 \leq p \&\& p \leq \text{length}(w)) \Rightarrow (\text{size(indexesOf}(\text{substring}(w,1,p), "0")) - \text{size(indexesOf}(\text{substring}(w,1,p), "1")) < 2 \\
\&\& \text{size(indexesOf}(\text{substring}(w,1,p), "1")) - \text{size(indexesOf}(\text{substring}(w,1,p), "0")) < 2) \)
Alphabet: 0 1

Benchmark 31
Statement: The set of strings of 0s and 1s whose number of 0s is divisible by five and whose number of 1s is even.
Formula: \( \text{size(indexesOf}(w, "0")) \% 5 = 0 \&\& \text{isEven(size(indexesOf}(w, "1"))) \)
Alphabet: 0 1

Benchmark 32
Statement: Strings that have a 1 either two or three positions from the end. Or "all strings of 0s and 1s such that either the second or third position from the end has a 1"
Formula: \( \text{fromEnd}(w, 2) = '1' || \text{fromEnd}(w, 3) = '1' \)
Alphabet: 0 1

Benchmark 33
Statement: The set of strings containing ab as a substring.
Formula: \( \text{contains}(w,"ab") \)
Alphabet: a b

From [7]
Benchmark 34
Statement: The set of strings having at most one pair of consecutive a’s and at most one pair of consecutive b’s.
Formula: \( \text{size}(\text{indexesOf}(w, "aa")) \leq 1 \) \&\& \( \text{size}(\text{indexesOf}(w, "bb")) \leq 1 \)
Alphabet: a b

Benchmark 35
Statement: The set of strings whose length is divisible by 6.
Formula: \( \text{length}(w) \mod 6 = 0 \)
Alphabet: a b

Benchmark 36
Statement: The set of strings whose 5th last symbol (5th symbol from the end) is b.
Formula: \( \text{fromEnd}(w, 5) = 'b' \)
Alphabet: a b

Benchmark 37
Statement: w is a binary string containing both substrings 010 and 101
Formula: \( \text{contains}(w, "010") \) \&\& \( \text{contains}(w, "101") \)
Alphabet: 0 1

Benchmark 38
Statement: the set of strings in \((a)^*\) whose length is divisible by either 2 or 7.
Formula: \( \text{length}(w) \mod 2 = 0 \) \| \( \text{length}(w) \mod 7 = 0 \)
Alphabet: a

Benchmark 39
Statement: the set of strings x in 0, 1* such that \#0 in x is even and \#1 in x is a multiple of three.
Formula: \( \text{isEven}(\text{size}(\text{indexesOf}(w, '0'))) \) \&\& \( \text{size}(\text{indexesOf}(w, '1')) \mod 3 = 0 \)
Alphabet: 0 1

Benchmark 40
Statement: Consider the language consisting of all words that have neither consecutive a’s nor consecutive b’s
Formula: \( !\text{contains}(w, "aa") \) \&\& \( !\text{contains}(w, "bb") \)
Alphabet: a b

Benchmark 41
Statement: Draw a DFSA that rejects all words for which the last two letters match.
Formula: \( !(\text{fromEnd}(w, 1) = \text{fromEnd}(w, 2)) \)
Alphabet: 0 1
**Benchmark 42**
Statement: Draw a DFSA that rejects all words for which the first two letters match.
Formula: \(! (\text{atPos}(w, 1) = \text{atPos}(w, 2))\)
Alphabet: 0 1

**Benchmark 43**
Statement: x contains an even number of a's.
Formula: isEven(size(indexesOf(w, "a")))
Alphabet: a b

**Benchmark 44**
Statement: x contains an odd number of b's.
Formula: isOdd(size(indexesOf(w, "b")))
Alphabet: a b

**Benchmark 45**
Statement: x contains an even number of a's and an odd number of b's
Formula: isEven(size(indexesOf(w, "a"))) \&\& 
isOdd(size(indexesOf(w, "b")))
Alphabet: a b

**Benchmark 46**
Statement: x contains an even number of a's or an odd number of b's
Formula: isEven(size(indexesOf(w, "a"))) \| 
isOdd(size(indexesOf(w, "b")))
Alphabet: a b

**Benchmark 47**
Statement: w contains substrings 010 and 101
Formula: contains(w, "010") \&\& contains(w, "101")
Alphabet: 0 1

**Benchmark 48**
Statement: w does not contain substring 0110
Formula: !contains(w, "0110")
Alphabet: 0 1

**Benchmark 49**
Statement: w has an even number of 0's and an even number of 1's
Formula: isEven(size(indexesOf(w, "0"))) \&\& 
isEven(size(indexesOf(w, "1")))
Alphabet: 0 1
**Benchmark 50**
Statement: \( w \) has the same number of occurrences of 10 and 01
Formula: \( \text{size(indexesOf}(w, "10")) = \text{size(indexesOf}(w, "01")) \)
Alphabet: 0 1

---

*By Prof. Sanjit A. Seshia at UC Berkeley*

**Benchmark 51**
Statement: Consider the set of all binary strings where the difference between the number of 0s and the number of 1s is even.
Formula: \( \text{isEven}(\text{size(indexesOf}(w, "0")) - \text{size(indexesOf}(w, "1"))) \)
Alphabet: 0 1

**Benchmark 52**
Statement: Let \( L \) be the language of all strings over the alphabet \( a, b \) which does not contain the string \( aa \) and does not contain the string \( bb \) and has length at least 1.
Formula: \( \neg\text{contains}(w, "aa") \land \neg\text{contains}(w, "bb") \land \text{length}(w) \geq 1 \)
Alphabet: a b

**Benchmark 53**
Statement: Give an NFA recognizing all binary strings which both do not contain \( 11 \) as a substring and do not end with 1.
Formula: \( \neg\text{contains}(w, "11") \land \neg(\text{fromEnd}(w, 1) = '1') \)
Alphabet: 0 1

**Benchmark 54**
Statement: \( w \) contains an even number of 1s, an odd number of 0s, and no occurrences of the sub-string 10.
Formula: \( \text{isEven}(\text{size(indexesOf}(w, "1"))) \land \text{isOdd}(\text{size(indexesOf}(w, "0"))) \land \neg\text{contains}(w, "10") \)
Alphabet: 0 1

**Benchmark 55**
Statement: Consider the language \( L \) consisting of all binary strings with an even number of 1s that are not immediately preceded by a 0. For example, 011011 is in \( L \) because it has two 1s that are not immediately preceded by a 0, whereas 111 is not in \( L \) because it has three 1s that are not immediately preceded by a 0.
Formula: \( -- \)
Alphabet: 0 1
**By Prof. Jeffrey D. Ullman’s at Stanford**

**Benchmark 56**  
Statement: Write a regular expression for the language $L$, over alphabet \{0, 1, 2\}, such that every 0 that is not the last (rightmost) symbol is immediately followed by a 1, and every 1 that is not the last symbol is immediately followed by a 0. By "immediately followed" we mean "with no intervening symbols. Thus, 010 is in $L$, but 001 is not, because the first 0 is not immediately followed by 1. 021 is not in $L$ for the same reason.

Formula:  
$$\forall p. (1 \leq p \land p < \text{length}(w)) \Rightarrow ((\in(p, \text{indexesOf}(w, "0")) \Rightarrow \text{atPos}(w, p+1) = '1') \land (\in(p, \text{indexesOf}(w, "1")) \Rightarrow \text{atPos}(w, p+1) = '0'))$$

Alphabet: 0 1 2

**By Dr. Mary Eberlein’s at UT Austin**

**Benchmark 57**  
Statement: The set of all binary strings having a substring 00 and ending with 01.

Formula:  
$$\text{contains}(w, "00") \land \text{isSuffix}("01", w)$$

Alphabet: 0 1

**Benchmark 58**  
Statement: The set of all binary strings having a substring 00 but not ending with 01.

Formula:  
$$\text{contains}(w, "00") \land \neg \text{isSuffix}("01", w)$$

Alphabet: 0 1

**Benchmark 59**  
Statement: The set of all binary strings with 3 consecutive zeros.

Formula:  
$$\text{contains}(w, "000")$$

Alphabet: 0 1

**Benchmark 60**  
Statement: The set of all binary strings beginning with a 1 which, interpreted as the binary representation of an integer, is congruent to 0 modulo 5.

Formula:  
Alphabet:

**Benchmark 61**  
Statement: The set of all strings over alphabet a, b of length up to 3.

Formula:  
$$\text{length}(w) \leq 3$$

Alphabet: a b
Benchmark 62
Statement: The set of all strings of length 3n, n = 0, 1, 2, ....
Formula: \(\text{length}(w) \% 3 = 0\)
Alphabet: 0 1

Benchmark 63
Statement: The set of all strings of 0s and 1s such that 10th symbol from the right end is a 1.
Formula: \(\text{fromEnd}(w, 10) = '1'\)
Alphabet: 0 1

Benchmark 64
Statement: \(\{w \in \{0,1\}^* \mid w\ \text{begins with a 1 and ends with a 0}\}\)
Formula: \(\text{atPos}(w, 1) = '1' \&\& \text{fromEnd}(w, 1) = '0'\)
Alphabet: 0 1

Benchmark 65
Statement: \(\{w \in \{0,1\}^* \mid w\ \text{contains at least 3 1s}\}\)
Formula: \(\text{size}(\text{indexesOf}(w, '1')) >= 3\)
Alphabet: 0 1

Benchmark 66
Statement: \(\{w \in \{0,1\}^* \mid w\ \text{starts with 0 and has odd length, or w starts with 1 and has even length}\}\)
Formula: \((\text{atPos}(w, 1) = '0' \&\& \text{isOdd}(\text{length}(w))) \mid\mid
(\text{atPos}(w,1)='1' \&\& \text{isEven}(\text{length}(w)))\)
Alphabet: 0 1

Benchmark 67
Statement: \(\{w \in \{0,1\}^* \mid w\ \text{contains at least two 0s and at most one 1}\}\)
Formula: \(\text{size}(\text{indexesOf}(w, "0")) >= 2 \&\& \text{size}(\text{indexesOf}(w, "1")) <= 1\)
Alphabet: 0 1

Benchmark 68
Statement: \(\{w \in \{0,1\}^* \mid \text{every odd position of } w\ \text{is a 1}\}\)
Formula: \(\forall p. (1 <= p \&\& p <= \text{length}(w) \&\& \text{isOdd}(p)) \Rightarrow \text{atPos}(w, p) = '1'\)
Alphabet: 0 1

Benchmark 69
Statement: The set of all strings ending in 00.
Formula: \(\text{isSuffix}("00", w)\)
Alphabet: 0 1
Benchmark 70
Statement: The set of all strings such that the 4th symbol from the right is 1.
Formula: \( \text{fromEnd}(w, 4) = '1' \)
Alphabet: 0 1

Benchmark 71
Statement: W contains at least 3 0s
Formula: \( \text{size}(\text{indexesOf}(w, "0")) \geq 3 \)
Alphabet: 0 1

Benchmark 72
Statement: Every odd position of w is a 1.
Formula: \( \forall p. (1 \leq p \land p \leq \text{length}(w) \land \text{isOdd}(p)) \implies \text{atPos}(w, p) = '1' \)
Alphabet: 0 1

Benchmark 73
Statement: W contains at least two a’s and at most one b.
Formula: \( \text{size}(\text{indexesOf}(w, 'a')) \geq 2 \land \text{size}(\text{indexesOf}(w, 'b')) \leq 1 \)
Alphabet: a b

Benchmark 74
Statement: For all prefixes of w, if \(|v| > 0 \) and \(|v| \) is even, then the last character of v is 1.
Formula: \( \forall p \, v. (1 \leq p \land p \leq \text{length}(w) \land \text{isEven}(\text{length}(v))) \implies \text{fromEnd}(v, 1) = '1' \)
Alphabet: 0 1

Benchmark 75
Statement: If there are any a’s in w, then there is at least one c, and if there are any b’s in w, there is at least one d.
Formula: \( \text{contains}(w, 'a') \implies \text{contains}(w, 'c') \land \text{contains}(w, 'b') \implies \text{contains}(w, 'd') \)
Alphabet: a b c d

Benchmark 76
Statement: w starts with the string 10 or ends with the string 01
Formula: \( \text{isPrefix}("10", w) \lor \text{isSuffix}("01", w) \)
Alphabet: 0 1
Benchmark 77
Statement: Binary strings in which every substring 010 is followed immediately by substring 111.
Formula: \( \forall p. \text{in}(p, \text{indexesOf}(w, "010")) \Rightarrow \text{substring}(w, p+3, p+6) = "111" \)
Alphabet: 0 1

By Prof. Jean Gallier at University of Pennsylvania

Benchmark 78
Statement: w has neither aa nor bb as a substring.
Formula: \( \neg \text{contains}(w, "aa") \&\& \neg \text{contains}(w, "bb") \)
Alphabet: a b

Benchmark 79
Statement: w has an even number of a's and an odd number of b's.
Formula: \( \text{isEven(size(indexesOf(w,"a"))) \&\& isOdd(size(indexesOf(w,"b")))} \)
Alphabet: a b

By Prof. Robert Sedgewick and Dr. Kevin Wayne at Princeton University

Benchmark 80
Statement: Draw a 4-state DFA that accepts the set of all bitstrings ending with 11. (Each state represents whether the input string read in so far ends with 00, 01, 10, or 11.) Draw a 3-state DFA that accomplishes the same task.
Formula: \( \text{isSuffix("11", w)} \)
Alphabet: 0 1

Benchmark 81
Statement: Draw a DFA for bitstrings with at least one 0 and at least one 1.
Formula: \( \text{contains}(w, '0') \&\& \text{contains}(w, '1') \)
Alphabet: 0 1

Benchmark 82
Statement: Draw an NFA that matches all strings that contain either a multiple of 3 1's or a multiple of 5 1's. Hint: Use 3 + 5 + 1 = 9 states and one epsilon transition.
Formula: \( \text{size(indexesOf(w, "1"))} \% 3 = 0 \mid\mid \text{size(indexesOf(w, "1"))} \% 5 = 0 \)
Alphabet: 0 1
Benchmark 83
Statement: Draw an NFA that recognize the language of all strings that end in aaab.
Formula: $\text{isSuffix}(\text{"aaab"}, w)$
Alphabet: a b

Benchmark 84
Statement: Draw an NFA that recognize the language of all strings whose 4th to the last character is a.
Formula: $\text{fromEnd}(w, 4)='a'$
Alphabet: a b

Benchmark 85
Statement: Draw an NFA that recognize the language of all strings whose 5th to the last character is a.
Formula: $\text{fromEnd}(w, 5)='a'$
Alphabet: a b

By Dr. Elaine Rich at UT Austin

Benchmark 86
Statement: w does not end in ba
Formula: $\neg \text{isSuffix}(\text{"ba"}, w)$
Alphabet: a b

Benchmark 87
Statement: w does not have 001 as a substring.
Formula: $\neg \text{contains}(w, \text{"001"})$
Alphabet: 0 1

Benchmark 88
Statement: The set of binary strings with at most one pair of consecutive 0's and at most one pair of consecutive 1's.
Formula: $\text{size}(\text{indexesOf}(w, \text{"00"})) \leq 1 \&\& \text{size}(\text{indexesOf}(w, \text{"11"})) \leq 1$
Alphabet: 0 1

Benchmark 89
Statement: w contains exactly two occurrences of the substring aa.
Formula: $\text{size}(\text{indexesOf}(w, \text{"aa"})) = 2$
Alphabet: a b
Benchmark 90
Statement: w contains no more than two occurrences of the substring aa.
Formula: \( \text{size(indexesOf}(w, "aa") \leq 2 \)
Alphabet: a b

By Prof. Jeff Foster at University of Maryland

Benchmark 91
Statement: Give a regular expression for all binary numbers including the substring "101"
Formula: contains(w, "101")
Alphabet: 0 1

Benchmark 92
Statement: Give a regular expression for all binary numbers with an even number of 1's
Formula: \( \text{isEven(size(indexesOf}(w, "1")}) \)
Alphabet: 0 1

Benchmark 93
Statement: Give a regular expression for all binary numbers that do not include "000"
Formula: !contains(w, "000")
Alphabet: 0 1

Benchmark 94
Statement: Give a regular expression for all binary numbers that begin and end with the same digit.
Formula: \( \text{atPos}(w, 1) = \text{fromEnd}(w, 1) \)
Alphabet: 0 1

Benchmark 95
Statement: Give a NFA that only accepts binary numbers that include either "00" or "11"
Formula: contains(w, "00") || contains(w, "11")
Alphabet: 0 1

Benchmark 96
Statement: Give a NFA that only accepts binary numbers that include both "00" and "11"
Formula: contains(w, "00") && contains(w, "11")
Alphabet: 0 1
By Goutam Biswas at IIT Kharagpur

Benchmark 97
Statement: Give a NFA that only accepts binary numbers such that $x \mod 3 = 0$ and $x$ is odd.
Formula: $\quad$
Alphabet: $0 \ 1$

Benchmark 98
Statement: Give a NFA that only accepts strings such that $x$ either has the substring $01$ or has the substring $021$.
Formula: $\text{contains}(w, \ "01") \mid \text{contains}(w, \ "021")$
Alphabet: $0 \ 1 \ 2$

Benchmark 99
Statement: Give the description of a DFA that recognizes all strings which have greater than or equal to 3 ‘a’s and less than or equal to 2 ‘b’s.
Formula: $\text{size}(\text{indexesOf}(w, \ "a")) \geq 3 \ \&\& \ \text{size}(\text{indexesOf}(w, \ "b")) \leq 2$
Alphabet: $a \ b$

By Prof. Tao Jiang at UC Riverside

Benchmark 100
Statement: Give the set of all strings that end with ”ing”
Formula: $\text{isSuffix}(\text{"ing"}, \ w)$
Alphabet: $a \ b \ c \ d \ e \ f \ g \ h \ i \ j \ k \ l \ m \ n \ o \ p \ q \ r \ s \ t \ u \ v \ w \ x \ y \ z$

Benchmark 101
Statement: Consider the DFA that accepts all strings which have 01 as a substring.
Formula: $\text{contains}(w, \ "01")$
Alphabet: $0 \ 1$

 Benchmark 102
Statement: Consider the DFA accepting all and only strings with an even number of 0’s and an even number of 1’s
Formula: $\text{isEven}(\text{size}(\text{indexesOf}(w, \ "0"))) \ \&\& \ 
\text{isEven}(\text{size}(\text{indexesOf}(w, \ "1")))$
Alphabet: $0 \ 1$
By Dr. Timo Kötzing at University of Delaware

**Benchmark 103**

Statement: Give the DFA M such that the language of M is \(ab, ba\)
Formula: \(\text{in}(w, "ab", "ba")\)
Alphabet: a b

**Benchmark 104**

Statement: Give a DFA such that it contains all strings that have "aba" as a substring
Formula: \(\text{contains}(w, "aba")\)
Alphabet: a b

### E.2 Benchmarks Statements (non-regular languages)

We now show the examples for the non-regularity proof construction that can also be translated into the UFPDL. We also try our automaton synthesis method on those examples. What happens in those cases is that the timeout of the UF-PDL solver (hence equivalence check) will bound the language. Since bounded languages are regular the system will return an (incorrect) automaton. In all cases the returned automata are larger than 10 states and the timeout is reached, so we know that the assumption of a small automaton is not respected. The most striking difference compared to regular language is the size of the counterexamples. When the counterexamples are small for regular languages, they are much larger for non-regular languages. The median counterexample size is 3 for regular examples and it becomes 16 for the non-regular cases.

**Benchmark 1**

Statement: A language with words with equal number of 0's and 1's
Formula: \(\text{size}(\text{indexesOf}(w, "0")) = \text{size}(\text{indexesOf}(w, "1"))\)
Alphabet: 0 1

**Benchmark 2**

Statement: A language with words with more number of 0's than 1's
Formula: \(\text{size}(\text{indexesOf}(w, "0")) > \text{size}(\text{indexesOf}(w, "1"))\)
Alphabet: 0 1

**Benchmark 3**

Statement: A language with words with equal number of 0's and 1's and 0's are before 1's
Formula: \(\text{match}(0^n1^m, w, \text{true})\)
Alphabet: 0 1
Benchmark 4
Statement: A language with words of form $0^m1^n0^m+n$
Formula: $\text{match}(0^m1^n0^l, w, m + n = l)$
Alphabet: $0 1$

Benchmark 5
Statement: A language with words of form $ww^r$
Formula: $\text{isEven(length(w))} \land (\forall p. (1 \leq p \land p \leq \text{length}(w)) \Rightarrow ((\text{atPos}(w, p) = \text{atPos}(w, (\text{length}(w) + 1 - p))))$
Alphabet: $0 1$

Benchmark 6
Statement: A language with words of form $w^r$ w
Formula: $\text{isEven(length(w))} \land (\forall p. (1 \leq p \land p \leq \text{length}(w)) \Rightarrow ((\text{atPos}(w, p) = '0') = (\text{atPos}(w, \text{length}(w) + 1 - p) = '1'))$
Alphabet: $0 1$

Benchmark 9
Statement: A language with words of form w = rev(w)
Formula: $\forall p . (1 \leq p \land p \leq \text{length}(w)) \Rightarrow \text{atPos}(w, p) = \text{atPos}(w, (\text{length}(w) + 1) - p)$
Alphabet: $0 1$

Benchmark 11
Statement: A language with words of the form $0^n1^n2^n$
Formula: $\text{match}(0^n1^n2^n, w, \text{true})$
Alphabet: $0 1 2$

Benchmark 12
Statement: A language with words with equal number of 0's and 1's
Formula: $\text{match}(0^a1^b0^k, w, (a > b) \land (b >= 0))$
Alphabet: $0 1$

Benchmark 13
Statement: A language with words of the form $0^n1^n2^n$
Formula: $\text{match}(0^n1^n2^n, w, \text{true})$
Alphabet: $0 1 2$

Benchmark 15
Statement: A language with words of the form $0^n1^m0^n$
Formula: $\text{match}(0^n1^m0^n, w, \text{true})$
Alphabet: $0 1$
Benchmark 16
Statement: A language without the words with equal number of 0's and 1's and 0's are before 1's
Formula: \(!\text{match}(0^n1^n, w, \text{true})\)
Alphabet: 0 1

Benchmark 17
Statement: A language with words with equal number of 0's and 1's and 0's are before 1's
Formula: \(\text{match}(0^n1^n, w, n \neq m)\)
Alphabet: 0 1

Benchmark 18
Statement: A language with words of form \(w \neq \text{rev}(w)\)
Formula: \(!\left(\forall p . (1 \leq p \&\& p \leq \text{length}(w)) \Rightarrow \text{atPos}(w, p) = \text{atPos}(w, (\text{length}(w) + 1) - p)\right)\)
Alphabet: 0 1

Benchmark 19
Statement: A language with words with equal number of 0's and 1's
Formula: \(\text{match}(0^j1^k2^l, w, j \neq 1 \&\& k = 1)\)
Alphabet: 0 1 2

E.3 Results

Tables 4 to 6 present the result of the experiments for the learning algorithm. The tables report the problem ID, the number of states of the resulting automaton, the longest counterexample used during the learning, whether the test was successful, and the time taken by the different methods. Unless explicitly specified, the times are given in milliseconds.

The example 38 that fails is a “counting” language. It accepts strings whose length is a multiple of 2 or 7. Solving this example requires increasing the bound from 12 to 14 states. On the other hand, our tool is able to learn correctly an example with 2047 states which is beyond our expectations. This example was in fact designed as an exercise for NFA rather than DFA construction. Thus, the NFA solution is small (12 states) which means short counterexamples are sufficient to learn it. However, the conversion to DFA leads in the worst-case to an exponential blowup. Nevertheless, a small NFA solution implies that short counterexamples are sufficient to learn the language.

With the previous observation in mind we investigate further the relation between the size of the counterexamples and the size of the automaton. In Figure 10 we show the distribution of the difference between the counterexample size and the automata size. With the knowledge of Proposition 2 we can expect the counterexamples to be about at most the automaton sizes 12. To our sur-
prise the counterexamples tend to be shorter in average than the automaton. From this experiment we can draw the conclusion that the assumption of small automaton in the context of education is mostly respected. Furthermore, it is possible to learn such automata using short traces.

![Distribution of the difference between the counterexample lengths and the automaton sizes](image)

Fig. 10: Distribution of the difference between the counterexample lengths and the automaton sizes

*Trying to learn non-regular languages.* Tables 7 presents what happens when we give non-regular languages to our tool. Those examples follow the pattern of producing large automata, using large counterexamples, and timing out. This pattern is the indication that the solution may be wrong.

**F  Detail of the Problem Generation Experiments**

Table ?? describes some alternates (i.e., a set of replacements for each operator/operand in that set) that can be used to convert a PDL formula into a template.

We have implemented the problem generation on top of our learning algorithms. The implementation of the problem generation decides when to replace functions by comparing the type signatures of the functions occurring on the seed problem. In most cases it matches Table ??, One notable difference is that our implementation does not distinguish between position and integer which leads to the creation of many invalid problems. The learning phase will remove some of the invalid problems, others result in trivial automaton. A failure in the learning in general corresponds to invalid positions. To determine if an automaton is trivial we check if it is empty or universal. Finally, the automata are grouped into equivalence classes. Two automata are considered similar if they have the same graph structure, i.e. we remove all the labels from the edges and the nodes and we check for isomorphism of the obtained graphs.

Table 9 presents the results. Due to the exponential nature of the generation we excluded the examples that leads to very large numbers of problems and the examples that have high learning time. This reduces the number of seed
| ID (1) | | S | | | | | | ex | | cex | | OK | | OK | | time | | Alg 0 time | | Alg 1 time | | Alg 2 time | |
| 1 | 3 | 2 | ✓ | 11806 | ✓ | 7276 | | | | | | | | | | | | | | | | |
| 2 | 5 | 2 | ✓ | 3075 | ✓ | 1609 | | | | | | | | | | | | | | | | |
| 3 | 4 | 2 | ✓ | 1852 | ✓ | 1529 | | | | | | | | | | | | | | | | |
| 5 | 7 | 2 | ✓ | 15048 | ✓ | 2176 | | | | | | | | | | | | | | | | |
| 6 | 4 | 3 | ✓ | 618 | ✓ | 571 | | | | | | | | | | | | | | | | |
| 7 | 4 | 3 | ✓ | 1178 | ✓ | 1074 | | | | | | | | | | | | | | | | |
| 8 | 3 | 2 | ✓ | 753 | ✓ | 1036 | | | | | | | | | | | | | | | | |
| 9 | 8 | 3 | ✓ | 30000+ | ✓ | 30000+ | | | | | | | | | | | | | | | | |
| 10 | 3 | 2 | ✓ | 520 | ✓ | 789 | | | | | | | | | | | | | | | | |
| 11 | 4 | 2 | ✓ | 1143 | ✓ | 1126 | | | | | | | | | | | | | | | | |
| 12 | 3 | 2 | ✓ | 285 | ✓ | 400 | | | | | | | | | | | | | | | | |
| 13 | 4 | 3 | ✓ | 609 | ✓ | 571 | | | | | | | | | | | | | | | | |
| 14 | 4 | 3 | ✓ | 615 | ✓ | 569 | | | | | | | | | | | | | | | | |
| 15 | 16 | 5 | ✓ | 30000+ | ✓ | 30000+ | | | | | | | | | | | | | | | | |
| 16 | 1024 | 7 | ✓ | 30000+ | ✓ | 30000+ | | | | | | | | | | | | | | | | |
| 17 | 6 | 4 | ✓ | 8511 | ✓ | 1298 | | | | | | | | | | | | | | | | |
| 18 | 15 | 8 | ✓ | 30000+ | ✓ | 5597 | | | | | | | | | | | | | | | | |
| 19 | 2047 | 2 | ✓ | 30000+ | ✓ | 30000+ | | | | | | | | | | | | | | | | |
| 20 | 9 | 6 | ✓ | 30000+ | ✓ | 30000+ | | | | | | | | | | | | | | | | |
| 21 | 4 | 2 | ✓ | 30000+ | ✓ | 30000+ | | | | | | | | | | | | | | | | |
| 22 | 10 | 6 | ✓ | 30000+ | ✓ | 30000+ | | | | | | | | | | | | | | | | |
| 23 | 11 | 11 | ✓ | 30000+ | ✓ | 12089 | | | | | | | | | | | | | | | | |
| 24 | 4 | 2 | ✓ | 30000+ | ✓ | 30000+ | | | | | | | | | | | | | | | | |
| 26 | 5 | 6 | ✓ | 3303 | ✓ | 1162 | | | | | | | | | | | | | | | | |
| 27 | 5 | 4 | ✓ | 30000+ | ✓ | 30000+ | | | | | | | | | | | | | | | | |
| 28 | 5 | 5 | ✓ | 2441 | ✓ | 1939 | | | | | | | | | | | | | | | | |
| 29 | 4 | 3 | ✓ | 693 | ✓ | 602 | | | | | | | | | | | | | | | | |
| 30 | 4 | 3 | ✓ | 18826 | ✓ | 10838 | | | | | | | | | | | | | | | | |
| 31 | 10 | 4 | ✓ | 30000+ | ✓ | 4120 | | | | | | | | | | | | | | | | |
| 32 | 5 | 4 | ✓ | 2370 | ✓ | 1297 | | | | | | | | | | | | | | | | |
| 33 | 3 | 2 | ✓ | 294 | ✓ | 406 | | | | | | | | | | | | | | | | |

Table 4: 1st part of the tests, (1) reports the problem ID, (2) the number of state of the automaton, (3) the longest counterexample used by L*, (4,6) whether the test was successful, (5,7,8) how much time did it take.
<table>
<thead>
<tr>
<th>ID</th>
<th>$S$</th>
<th>OK</th>
<th>time</th>
<th>OK</th>
<th>time</th>
<th>OK</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>10</td>
<td>✓</td>
<td>33415</td>
<td>✓</td>
<td>3162</td>
<td>✓</td>
<td>60</td>
</tr>
<tr>
<td>35</td>
<td>6</td>
<td>✓</td>
<td>3280</td>
<td>✓</td>
<td>1023</td>
<td>✓</td>
<td>9</td>
</tr>
<tr>
<td>36</td>
<td>32</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>5919</td>
<td>✓</td>
<td>57</td>
</tr>
<tr>
<td>37</td>
<td>12</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>2636</td>
<td>✓</td>
<td>27</td>
</tr>
<tr>
<td>38</td>
<td>2</td>
<td>✓</td>
<td>21</td>
<td>×</td>
<td>21</td>
<td>✓</td>
<td>38</td>
</tr>
<tr>
<td>39</td>
<td>6</td>
<td>✓</td>
<td>8074</td>
<td>✓</td>
<td>2207</td>
<td>✓</td>
<td>23</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>✓</td>
<td>1127</td>
<td>✓</td>
<td>1035</td>
<td>✓</td>
<td>17</td>
</tr>
<tr>
<td>41</td>
<td>5</td>
<td>✓</td>
<td>3426</td>
<td>✓</td>
<td>1171</td>
<td>✓</td>
<td>11</td>
</tr>
<tr>
<td>42</td>
<td>4</td>
<td>✓</td>
<td>2029</td>
<td>✓</td>
<td>1738</td>
<td>✓</td>
<td>8</td>
</tr>
<tr>
<td>43</td>
<td>2</td>
<td>✓</td>
<td>250</td>
<td>✓</td>
<td>370</td>
<td>✓</td>
<td>2</td>
</tr>
<tr>
<td>44</td>
<td>2</td>
<td>✓</td>
<td>250</td>
<td>✓</td>
<td>368</td>
<td>✓</td>
<td>2</td>
</tr>
<tr>
<td>45</td>
<td>4</td>
<td>✓</td>
<td>1802</td>
<td>✓</td>
<td>1452</td>
<td>✓</td>
<td>17</td>
</tr>
<tr>
<td>46</td>
<td>4</td>
<td>✓</td>
<td>1816</td>
<td>✓</td>
<td>1463</td>
<td>✓</td>
<td>7</td>
</tr>
<tr>
<td>48</td>
<td>5</td>
<td>✓</td>
<td>1428</td>
<td>✓</td>
<td>777</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>49</td>
<td>4</td>
<td>✓</td>
<td>1805</td>
<td>✓</td>
<td>1445</td>
<td>✓</td>
<td>7</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
<td>✓</td>
<td>410</td>
<td>✓</td>
<td>1442</td>
<td>✓</td>
<td>11</td>
</tr>
<tr>
<td>51</td>
<td>2</td>
<td>✓</td>
<td>416</td>
<td>✓</td>
<td>531</td>
<td>✓</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
<td>✓</td>
<td>1567</td>
<td>✓</td>
<td>1875</td>
<td>✓</td>
<td>7</td>
</tr>
<tr>
<td>53</td>
<td>3</td>
<td>✓</td>
<td>618</td>
<td>✓</td>
<td>644</td>
<td>✓</td>
<td>17</td>
</tr>
<tr>
<td>54</td>
<td>5</td>
<td>✓</td>
<td>4556</td>
<td>✓</td>
<td>2107</td>
<td>✓</td>
<td>32</td>
</tr>
<tr>
<td>56</td>
<td>4</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>1818</td>
</tr>
<tr>
<td>57</td>
<td>5</td>
<td>✓</td>
<td>1890</td>
<td>✓</td>
<td>920</td>
<td>✓</td>
<td>8</td>
</tr>
<tr>
<td>58</td>
<td>5</td>
<td>✓</td>
<td>5788</td>
<td>✓</td>
<td>978</td>
<td>✓</td>
<td>5</td>
</tr>
<tr>
<td>60</td>
<td>5</td>
<td>✓</td>
<td>1345</td>
<td>✓</td>
<td>760</td>
<td>✓</td>
<td>4</td>
</tr>
<tr>
<td>62</td>
<td>3</td>
<td>✓</td>
<td>397</td>
<td>✓</td>
<td>479</td>
<td>✓</td>
<td>3</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>✓</td>
<td>1161</td>
<td>✓</td>
<td>1060</td>
<td>✓</td>
<td>4</td>
</tr>
<tr>
<td>65</td>
<td>4</td>
<td>✓</td>
<td>1065</td>
<td>✓</td>
<td>1397</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>66</td>
<td>3</td>
<td>✓</td>
<td>986</td>
<td>✓</td>
<td>1238</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>67</td>
<td>7</td>
<td>✓</td>
<td>15974</td>
<td>✓</td>
<td>2415</td>
<td>✓</td>
<td>16</td>
</tr>
<tr>
<td>68</td>
<td>3</td>
<td>✓</td>
<td>5290</td>
<td>✓</td>
<td>5457</td>
<td>✓</td>
<td>10</td>
</tr>
<tr>
<td>70</td>
<td>16</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>3280</td>
<td>✓</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 5: 2nd part of the tests, for explanation of the column see Table 4.
<table>
<thead>
<tr>
<th>ID</th>
<th>S</th>
<th>OK</th>
<th>Alg 0 time</th>
<th>Alg 1 time</th>
<th>Alg 2 time</th>
</tr>
</thead>
<tbody>
<tr>
<td>71</td>
<td>4</td>
<td>✓</td>
<td>1061</td>
<td>✓</td>
<td>794</td>
</tr>
<tr>
<td>73</td>
<td>3</td>
<td>✓</td>
<td>15459</td>
<td>✓</td>
<td>2416</td>
</tr>
<tr>
<td>74</td>
<td>2</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>30000+</td>
</tr>
<tr>
<td>75</td>
<td>9</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>30000+</td>
</tr>
<tr>
<td>76</td>
<td>6</td>
<td>✓</td>
<td>3642</td>
<td>✓</td>
<td>1252</td>
</tr>
<tr>
<td>77</td>
<td>7</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>11366</td>
</tr>
<tr>
<td>80</td>
<td>3</td>
<td>✓</td>
<td>268</td>
<td>✓</td>
<td>390</td>
</tr>
<tr>
<td>81</td>
<td>4</td>
<td>✓</td>
<td>973</td>
<td>✓</td>
<td>936</td>
</tr>
<tr>
<td>82</td>
<td>15</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>5924</td>
</tr>
<tr>
<td>83</td>
<td>5</td>
<td>✓</td>
<td>1102</td>
<td>✓</td>
<td>680</td>
</tr>
<tr>
<td>84</td>
<td>4</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>2889</td>
</tr>
<tr>
<td>85</td>
<td>3</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>5772</td>
</tr>
<tr>
<td>86</td>
<td>2</td>
<td>✓</td>
<td>327</td>
<td>✓</td>
<td>425</td>
</tr>
<tr>
<td>87</td>
<td>4</td>
<td>✓</td>
<td>685</td>
<td>✓</td>
<td>601</td>
</tr>
<tr>
<td>88</td>
<td>6</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>3136</td>
</tr>
<tr>
<td>89</td>
<td>7</td>
<td>✓</td>
<td>8232</td>
<td>✓</td>
<td>1578</td>
</tr>
<tr>
<td>90</td>
<td>7</td>
<td>✓</td>
<td>8327</td>
<td>✓</td>
<td>1605</td>
</tr>
<tr>
<td>91</td>
<td>3</td>
<td>✓</td>
<td>586</td>
<td>✓</td>
<td>558</td>
</tr>
<tr>
<td>92</td>
<td>0</td>
<td>✓</td>
<td>252</td>
<td>✓</td>
<td>369</td>
</tr>
<tr>
<td>93</td>
<td>4</td>
<td>✓</td>
<td>691</td>
<td>✓</td>
<td>604</td>
</tr>
<tr>
<td>94</td>
<td>3</td>
<td>✓</td>
<td>1845</td>
<td>✓</td>
<td>1086</td>
</tr>
<tr>
<td>95</td>
<td>4</td>
<td>✓</td>
<td>1157</td>
<td>✓</td>
<td>925</td>
</tr>
<tr>
<td>96</td>
<td>8</td>
<td>✓</td>
<td>16467</td>
<td>✓</td>
<td>1683</td>
</tr>
<tr>
<td>98</td>
<td>4</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>30000+</td>
</tr>
<tr>
<td>99</td>
<td>6</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>4359</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>✓</td>
<td>30000+</td>
<td>✓</td>
<td>30000+</td>
</tr>
<tr>
<td>101</td>
<td>3</td>
<td>✓</td>
<td>303</td>
<td>✓</td>
<td>276</td>
</tr>
<tr>
<td>103</td>
<td>5</td>
<td>✓</td>
<td>1065</td>
<td>✓</td>
<td>3627</td>
</tr>
<tr>
<td>104</td>
<td>3</td>
<td>✓</td>
<td>592</td>
<td>✓</td>
<td>559</td>
</tr>
</tbody>
</table>

Table 6: 3rd part of the tests, for explanation of the column see Table 4.
### Table 7: Trying to learn non regular languages, for explanation of the column see Table 4.

<table>
<thead>
<tr>
<th>ID</th>
<th></th>
<th>cex</th>
<th>Alg 0</th>
<th>Alg 1</th>
<th>Alg 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
<td>22</td>
<td>OK</td>
<td>OK</td>
<td>19763</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>23</td>
<td>OK</td>
<td>OK</td>
<td>17529</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>18</td>
<td>OK</td>
<td>30000+</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td>12</td>
<td>OK</td>
<td>30000+</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>449</td>
<td>16</td>
<td>OK</td>
<td>30000+</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>420</td>
<td>16</td>
<td>OK</td>
<td>30000+</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>721</td>
<td>14</td>
<td>OK</td>
<td>30000+</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>7</td>
<td></td>
<td></td>
<td>30000+</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>11</td>
<td></td>
<td></td>
<td>30000+</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>9</td>
<td></td>
<td></td>
<td>30000+</td>
</tr>
<tr>
<td>15</td>
<td>19</td>
<td>11</td>
<td></td>
<td></td>
<td>30000+</td>
</tr>
<tr>
<td>16</td>
<td>19</td>
<td>18</td>
<td></td>
<td></td>
<td>30000+</td>
</tr>
<tr>
<td>17</td>
<td>15</td>
<td>11</td>
<td></td>
<td></td>
<td>30000+</td>
</tr>
<tr>
<td>18</td>
<td>624</td>
<td>14</td>
<td></td>
<td></td>
<td>30000+</td>
</tr>
<tr>
<td>19</td>
<td>11</td>
<td>7</td>
<td></td>
<td></td>
<td>30000+</td>
</tr>
</tbody>
</table>

### Table 8: Sample alternates for problem generalization at level of English and corresponding PDL descriptions.

<table>
<thead>
<tr>
<th>English</th>
<th>PDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>every, some, no</td>
<td>∃, ∀, ∀¬</td>
</tr>
<tr>
<td>and, or, if, if and only if</td>
<td>∧, ∨, ⇒, ⇔</td>
</tr>
<tr>
<td>not, ϵ</td>
<td>¬, ϵ</td>
</tr>
<tr>
<td>even, odd</td>
<td>isEven, isOdd</td>
</tr>
<tr>
<td>start with, end with, contain</td>
<td>isPrefix, isSuffix, contains</td>
</tr>
<tr>
<td>00, 01, 10, 11, 000, 001...</td>
<td>“00”, “01”, “10”, “11”, “000”, “001”...</td>
</tr>
<tr>
<td>equal, less than, not equal, greater than, less than or equal/at most, greater than or equal/at least</td>
<td>=, &lt;, ≠, &gt;, ≤, ≥</td>
</tr>
<tr>
<td>zero, one, two, three</td>
<td>0, 1, 2, 3</td>
</tr>
<tr>
<td>first, second, third, fourth</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>left/start, right/end</td>
<td>fromStart, fromEnd</td>
</tr>
</tbody>
</table>
problems to 52. Furthermore, when different problems gives corresponds to the same template, we keep only one of them which result in only 23 templates. The first column is the ID of the problem used as a seed. Column (2) shows the number of generated problems. Column (3) shows how many problems we were able to lean. Column (4) show the number of non-trivial problems (not empty or not universal) (5) show the number of equivalence classes when using the automata graph structure as similarity criterion.

The median number of generated problems is 400. The median number of learned problems is 360. The median number of non-trivial problems is 304. The median number of equivalence classes is 12.

<table>
<thead>
<tr>
<th>ID</th>
<th>generated</th>
<th>learned</th>
<th>non-trivial</th>
<th>equiv. classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>640</td>
<td>640</td>
<td>416</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>150</td>
<td>150</td>
<td>144</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>64000</td>
<td>40960</td>
<td>38784</td>
<td>78</td>
</tr>
<tr>
<td>8</td>
<td>6400</td>
<td>5120</td>
<td>5120</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>13500</td>
<td>13500</td>
<td>9880</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>3200</td>
<td>2560</td>
<td>2560</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>70</td>
<td>70</td>
<td>64</td>
<td>7</td>
</tr>
<tr>
<td>17</td>
<td>49000</td>
<td>49000</td>
<td>40296</td>
<td>118</td>
</tr>
<tr>
<td>24</td>
<td>360</td>
<td>360</td>
<td>276</td>
<td>3</td>
</tr>
<tr>
<td>28</td>
<td>2700</td>
<td>2054</td>
<td>896</td>
<td>11</td>
</tr>
<tr>
<td>29</td>
<td>300</td>
<td>300</td>
<td>288</td>
<td>12</td>
</tr>
<tr>
<td>35</td>
<td>900</td>
<td>730</td>
<td>448</td>
<td>12</td>
</tr>
<tr>
<td>36</td>
<td>80</td>
<td>64</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>41</td>
<td>800</td>
<td>512</td>
<td>512</td>
<td>31</td>
</tr>
<tr>
<td>43</td>
<td>12</td>
<td>8</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>1440</td>
<td>640</td>
<td>416</td>
<td>3</td>
</tr>
<tr>
<td>48</td>
<td>620</td>
<td>620</td>
<td>608</td>
<td>28</td>
</tr>
<tr>
<td>81</td>
<td>160</td>
<td>160</td>
<td>104</td>
<td>3</td>
</tr>
<tr>
<td>83</td>
<td>310</td>
<td>310</td>
<td>304</td>
<td>29</td>
</tr>
<tr>
<td>86</td>
<td>140</td>
<td>140</td>
<td>128</td>
<td>7</td>
</tr>
<tr>
<td>92</td>
<td>12</td>
<td>8</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>94</td>
<td>400</td>
<td>256</td>
<td>256</td>
<td>32</td>
</tr>
<tr>
<td>103</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 9: Problem generation: (1) ID (2) variations generated (3) automata learned (4) non-trivial automata (5) equivalence classes of similar automata