University of Amsterdam

Master Thesis

Grammatical Inference from Source Code Written in an Unknown Programming Language

Author:
Ovidiu Roșu

Supervisor:
Dr. Vadim Zaytsev

A thesis submitted in fulfilment of the requirements for the degree of MSc Software Engineering

in the

Faculty of Science

June 27, 2014

UNIVERSITY OF AMSTERDAM

Abstract

Faculty of Science

MSc Software Engineering

Grammatical Inference from Source Code Written in an Unknown Programming Language

by Ovidiu Roșu

The most advanced and precise tools of source code analysis and manipulation are based on structural definitions of software languages (i.e., grammars). The first step in building such tools entails acquiring a grammar through extraction, recovery and other means. One of such means is inference, when a grammar is inferred automatically based on a corpus of source code examples and some method-specific assumptions. In this project, we have analysed several previously published methods of grammatical inference, reimplemented three of them and presented a detailed technical overview together with quantitative and qualitative results of their replication. This thesis also contains necessary background for the field of grammatical inference.

Contents

Abbreviations

APTA **A**ugmented **P**refix **T**ree **A**cceptor. 15–17, 19, 21–23, 25, 26, 28

BFG **B**inary **F**eature **G**rammar. 13

CCFG **C**ongruential **C**ontext-**F**ree **G**rammar. 33

CFG **C**ontext-**F**ree **G**rammar. 6, 9, 12–14, 31–33, 35–37, 41

CFL **C**ontext-**F**ree **L**anguage. 13, 14, 31, 33, 35

CNF **C**homsky **N**ormal **F**orm. 12, 33, 37

CYK **C**ocke **Y**ounger **K**asami. 9, 37

DFA **D**eterministic **F**inite **A**utomaton. 1, 14–17, 19, 20, 22, 25, 28, 31, 37, 41

DSL **D**omain-**s**pecific **l**anguage. 1

EDSM **E**vidence **D**riven **S**tate **M**erging. 16, 17, 19, 25–29, 37, 41

MAT **M**inimally **A**dequate **T**eacher. 11, 12, 35–37

NP **N**on-deterministic **P**olynomial-time. 3, 19

PAC **P**robably **A**pproximately **C**orrect. 3, 14

RHS **R**ight **H**and **S**ide. 7

Introduction

Software maintenance is a crucial part in every software project. According to studies and statistics, around 40 to 80% of a project's budget is spent on maintenance [1]. As software rapidly grows in size and complexity, the problem only becomes worse with time. Bug detection and removal are major tasks in the maintenance phase of the software life cycle. Multiple analysis tools for software fault localization are developed, and a grammar of a programming language provides substantial aid in building these software engineering tools.

The term **grammar** is used in the sense of all established grammar formalisms and grammar notations including context-free grammars, class dictionaries, XML schemas as well as some forms of tree and graph grammars [2]. Grammar knowledge plays an important role in multiple software development scenarios. These scenarios include: engineering program analysis and modification tools, testing based on grammar/DFA, converging the inferred grammar with the one provided by the language developers to assess language use, structural pattern recognition, information retrieval, Web mining, text processing, and extending/redisigning of existing DSLs.

Sometimes, the grammars of languages are not available or they are incomplete, and have to be inferred from source code; especially in the case of programming language dialects. **Grammatical inference** (or grammar induction) is the process of automatically inferring a grammar by examining the sentences of an unknown language [3]. Grammatical inference is a mature topic and there are multiple published papers, that treat techniques and algorithms on how to induct grammars of different types of languages. These methods are tested on small samples of data and none of them (so far) provide solid evidence of inferring a grammar of a mainstream programming language from source code.

The goal of this thesis is to find a published inference method / algorithm, that might be used to infer a grammar of a complex programming language from source code. Stevenson et al. introduce the theory of grammatical inference, and present a systematic exploration of inference techniques in software engineering [3]. Our approach is to take it further and implement some of the methods, analyze applicability and report on it.

Chapter 1 covers the basic notions of grammars along with brief descriptions of different learning models. Chapter 2 comprises a survey of inference methods for programming languages. In chapter 3, we explore two state of the art algorithms for finding minimal DFAs, and we investigate if they can be practically used for inferring a grammar from source code. Chapter 4 describes a method of inferring a context-free grammar from positive samples using a Minimally Adequate Teacher.

Chapter 1

Basic Notions of Grammars and Language Learning

1.1 Learning Models

In order to investigate the plausibility and limitations of an inference problem, it is important to know the type of learning model used by the inference method. This section briefly describes the characteristics of the main learning models used in grammatical inference. The classification is borrowed from the paper written by Stevenson et al. [3].

1.1.1 Identification in the limit

The first learning model used in grammatical inference was introduced by Gold in 1967 [4]. In this paper, Gold addressed the following question: "Is the information sufficient to determine which of the possible languages is the unknown language?". He showed that an inference algorithm can identify an unknown language in the limit from complete information in a finite number of steps.

The input of this method is a sequence of strings (known as presentation or information). The more samples that are used as input to the inference algorithm, the better is the approximation of the target language. Given a sufficiently large sequence of strings, the algorithm converge on the target language.

There are two types of presentation: positive presentation and complete presentation. In the first type, the input strings are always part of the targeted language while in the second type, the input sequence also contains strings that are not part of the targeted language and are marked as such.

The problem with this method is that the inference algorithm will not know when it has identified the correct language, because there is always a chance the next sample will invalidate the last hypothesis. Angluin offers a possible way to avoid this problem through "tell-tales" [5], an unique set of strings that makes the difference between languages.

1.1.2 Teacher and Queries

This method uses a teacher (also known as an oracle), who knows the target language and can answer particular types of questions (queries) from the inference algorithm. The teacher is often a human.

Angluin describes six types of queries, two of which have a significant impact on language learning: membership and equivalence queries [6]. For the membership queries, the inference algorithm presents a string to the oracle and receives a "yes" or "no" answer. For the equivalence queries, the inference algorithm presents a grammar hypothesis to the oracle and receives "yes" if the the hypothesis is true and "no" otherwise. A teacher who can answer both types of queries is known as minimally adequate teacher.

1.1.3 Probably Approximately Correct Learning

Valiant proposed a learning model that has elements of both "identification in the limit" and "teachers and queries", the Probably Approximately Correct (PAC) Learning Model [7]. It also differs from those two because it does not guarantee exact identification with certainty, it is a compromise between accuracy and certainty.

The problem with this method is that the inference algorithm must learn in polynomial time under all distributions, but it is believed to be too strict to be practical. The reason for that is that many apparently simple concept classes are either known to be NP-hard, or at least not known to be polynomially learnable for all distributions [3]. To mitigate this problem, Li et al. propose to alter the inference algorithm to consider only simple distributions [8].

1.2 Complexity

The literature in the field of grammatical inference is mostly dedicated to the analysis of learnability and complexity of different language classes. The results typically state if a language class is learnable or not, consider techniques of learning in polynomial time, find the minimal grammar for a target language or identify the language with a particular probability.

Gold showed that a large class of languages can be identified in the limit, by using a style similar to brute-force called "identification by enumeration" [4]. By using complete information, this style can identify regular, context-free, context-free, context-sensitive and primitive recursive classes.

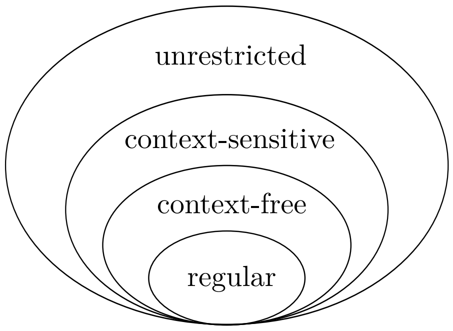


FIGURE 1.1: Chomsky hierarchy (source)

In 1956, Chomsky introduced a hierarchy of classes of formal grammars [9], that later became known as Chomsky hierarchy.

This hierarchy consists of the following levels:

- Type-0 grammars (unrestricted grammars) include all formal grammars
- Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages
- Type-2 grammars (context-free grammars) generate the context-free languages
- Type-3 grammars (regular grammars) generate the regular languages

The broader the class of the language, the harder it is to process and develop methods for it. We start our search with the smallest class (regular) and go further from there. However, regular languages have their limitations and might not help us, and that is the reason why we also investigate the next smallest class (context-free), which covers most of the cases.

Context-free grammars are the theoretical basis for the phrase structure of most programming languages. Regular grammars define regular languages, which are commonly used to define search patterns. These two grammar classes, which are the easiest classes in Chomsky hierarchy (type-2 and type-3), are analyzed in the following chapters.

Chapter 2

Survey of Inference Methods for Programming Languages

In this chapter we briefly describe some grammatical inference methods, and analyze their applicability to programming languages. We also examine the cases in which the methods may be useful, and also the assumptions on which the methods are built upon. Each section of this chapter is structured as follows: a small introduction, a list of assumptions, a brief description of the method and finally, some implementation details.

2.1 Gramin: A System for Incremental Learning of Programming Language Grammars

Many software vendors provide software maintenance as a service. They often rely on analysis tools, developed in-house, for software fault localization. Many of these tools are grammar-based, and therefore, a programming language grammar is required. The main differences between learning a grammar in programming language domains and other domains are the bigger size of samples, and the complexities of the programming language grammar [10].

Saha et al. propose a programming language grammar inference system, called **Gramin**, which is used to infer a grammar from sample programs [10]. Gramin employs various optimizations to make the inference process more practical for programming languages.

2.1.1 Assumptions

- we have a large number of programs

- we have an initial set of grammar rules (an incomplete CFG)
- we have a lexer which is already in its correct form
- we know at least one rule for each type of statement
- we do not have access to the compiler for the language to be learned

2.1.2 Algorithm

Gramin starts by breaking the input programs into consistent statements using the lexer. For each type of statement, Gramin employs the learning algorithm from its positive sample set, in order of decreasing frequency. The pseudo-code and more examples can be found in [11].

Gramin employs an iterative technique (called $Gramin - one$), in which each iteration takes one sample as input, determines a set of rules that along with input rules parses the given sample [10]. If for any sample, it cannot determine any rule that parses the sample, then the process backtracks to the last sample, and finds another set of rules.

Because there can be multiple sets of rules, Gramin uses various goodness criteria to choose a set of rules over the other. One of these optimizations is the $Focus$ algorithm, which reduces the scope of search of rules in $Gramin - one$ to those nonterminals where the change can occur. This optimization is based on the observation that only a few tokens are responsible for unsuccessful parsing of a string. Saha et al. analyze and present the effects of the optimizations employed by Gramin [10].

2.1.3 Implementation

Saha et al. showed the results of some of the experiments performed using Gramin [10], implemented in $Java$ and XSB $Prolog$. Because XSB uses a top-down goal directed resolution strategy, Gramin does not compute the entire bridge relation. Instead, it only generates the result sets obtained by the focus algorithm.

The presented case study is a complex industry standard language called $ABAP$. Because the language is proprietary, the grammar is not publicly available. In their experiments, they start with a basic grammar that can parse only the first statement of this language, and describe how the grammar changes with every iteration. They also show the effectiveness of some of the optimizations compared with an unoptimized version.

We asked the authors of this method to share the source code, so we could test it and benchmark it with multiple samples. One of the authors answered saying that they are not allowed by IBM to share source code.

2.2 Inferring Grammar Rules of Programming Language Dialects

The grammar of a programming language is an important asset because it is used in building many software engineering tools. Sometimes, grammars of languages are not available and have to be inferred from source code; especially in the case of programming language dialects.

Dubey et al. propose a method for inferring the grammar of a programming language when an incomplete grammar along with a set of correct programs is given as input [12]. The authors of this paper also suggest a rule evaluation order (the order in which the rules are evaluated for correctness) to improve the process of grammar inference.

Their experiments show that the rule evaluation order improves the performance of the inference algorithm [12]. This rule evaluation order is extended and explained by Dubey in more detail in another paper titled "Goodness Criteria for Programming Language Grammar Rules" [13].

2.2.1 Assumptions

- we have a set of correct programs
- we have an incomplete grammar
- the statements corresponding to each new keyword can be expressed by exactly one grammar rule
- the set of new keywords is known beforehand (the lexical analyzer does not recognize a new keyword as an existing terminal or as an unknown terminal)
- the right hand side (RHS) of missing rules start with a new keyword

2.2.2 Algorithm

The algorithm proposed by Dubey et al. uses an iterative approach with backtracking [12]. In each iteration, a set of possible rules corresponding to a new terminal is built and one among them is added in the grammar. The initial version of the method receives as input an incomplete grammar with one rule missing which, if found, will make the grammar complete.

If the grammar requires multiple rules to be found, in order to become complete, the algorithm iteratively builds a set of possible rules corresponding to each new terminal and adds them in the grammar. This method requires that there is at least one program where a new terminal K is the last new terminal. Once a rule corresponding to K is added in the grammar, K no longer remains a new terminal.

Because the rule evaluation order has a huge influence on the algorithm's performance, Dubey et al. propose a criterion for rule evaluation order [12]. This criterion is tested with different programming language grammars: viz, C, $Java$, $Matlab$, $Cobol$. They discovered that the rules with higher weight criterion are generally correct rules (in almost 97% of the cases).

2.2.3 Implementation

The authors presented the results of their experiments for four programming languages: C, $Java$, $Matlab$ and $Cobol$. These experiments were conducted only on a few statement types (like $case$, $switch$, $enum$, $while$, for) and time measurements are shown with and without the optimizations. The same method is tested on grammars of the same programming languages, in which multiple rules are missing, and again, these grammars define only a few statement types.

We asked the authors of this method to share the source code, so we could test it and benchmark it. Unfortunately, we did not receive any response so far.

2.3 Learning context-free grammar rules from a set of programs

The grammar of a programming language is important because it can be used to develop efficient program analysis and modification tools. Sometimes programs are written in dialects and the grammars of these dialects may not be available.

Dubey et al. propose a technique for reverse engineering context-free grammar rules from a given set of programs. Using an iterative approach with backtracking, an approximate grammar is generated [14]. The authors explain the correctness of this approach and propose a set of optimisations to make it more efficient.

2.3.1 Assumptions

- we have a set of correct programs
- we have an approximate grammar (G') for which we know the new terminals
- a single rule is sufficient to express the syntactic construct corresponding to each new terminal
- the types of missing rules are constructs that involve new keywords, operators or declaration specifiers
- additional rules in the grammar to be learned (G') are of the form $A \rightarrow \alpha a_{\text{new}} \beta$, where A is a nonterminal, a_{new} is a new terminal and $\alpha, \beta \in (N \cup T)^*$

2.3.2 Algorithm

Dubey et al. propose an iterative technique with backtracking [14]. A set of possible rules is built in each iteration and one of them is added to the grammar. After a certain number of iterations, the modified grammar is checked to see whether it parses all the programs or not. If the grammar does not parse all the programs, the algorithm backtracks to the previous iteration and selects another rule.

At first, Dubey et al. present a method for inferring a single correcting rule from a set of programs and an approximate grammar. They assume that there exists at least one complete CFG that is just one rule away from the initial grammar. After that, they extend this method for multiple correcting rules. The authors propose an iterative algorithm with backtracking. In each iteration, a set of possible rules corresponding to a new terminal is built [14].

The authors provide a proof of correctness for the described method for those situations when a single rule is sufficient to complete the grammar, and show how the proof can be extended for multiple rules. They also provide some optimizations to make the inference algorithm practical for programming languages. The goal of these optimizations is to reduce the number of possible rules by utilising unit productions, or use a modified CYK parsing algorithm in the rule checking process.

2.3.3 Implementation

Dubey et al. implemented the algorithm in Java and they presented the results of it over four programming languages: *Java*, *C*, *MATLAB* and *Cobol*. Apart from that, they also performed experiments on a real dialect of C, which is C^*. The experiments were conducted on a few statement types: *for*, *while*, *switch*, *case*, *break* and the like. The approach does not cover all types of extensions that could happen in programming languages.

We asked the authors of this method to share the source code, so we could test it and benchmark it. Unfortunately, we did not receive any response so far.

2.4 Example-Driven Reconstruction of Software Models

Certain reverse engineering methods attempt to build software models from source code with the help of a suitable parser for that language. Unfortunately, if the parser is not available, a lot of effort needs to be invested in building it [15]. On the other hand, a complete parser (that recognizes all language constructs) is not typically required and we usually have access to legacy examples.

Nierstrasz et al. propose an approach that uses the observations specified above to rapidly and incrementally develop parsers [15]. This method follows a simple set of steps: specify mappings from source code examples to model elements; use the mappings to generate a parser; parse as much code as we can; use exceptional cases to develop new example mappings; and then iterate.

2.4.1 Assumptions

- we have good knowledge of the language to be learned
- we have correct programs written in the language to be learned
- we have a simple, high-level interface for specifying mapping from example code to model elements

2.4.2 Algorithm

Nierstrasz et al. propose an approach that starts by specifying a set of mappings from source code examples to model elements [15]. These mappings are used to automatically generate a parser for the specified examples, that will directly produce model elements. The parser is applied to the source code. The code pieces that cannot be parsed are flagged and are used to specify new mappings for specified examples. Adding these new mappings, a new parser is generated. This parser tries to parse the remaining code pieces. The process is repeated until all, or enough of the code is analyzed. "Enough" in this case is dependent on the reverse engineering goal, e.g. when we have enough information for a specific reverse engineering task.

The lexical analysis is performed by a simple, generic scanner that breaks source code files into streams of tokens (identifiers, numbers, comments and the like). After that, the engineer specifies how these tokens map to the target model elements and constructs signatures for each case. In the next step, the engineer traverses the example tree and generates a grammar production for each node that maps to a target element of the meta-model [15]. From the generated grammar, a parser is automatically constructed. This parser will build model elements from the parsed code.

2.4.3 Implementation

In order to test this approach, Nierstrasz et al. built a proof-of-concept prototype, called **CodeSnooper**, that uses the example-driven model reconstruction. They applied CodeSnooper to two different case studies: Java and Ruby. The results of their experiments can be seen in the original paper [15], as well as in the MSc Thesis of Kobel [16].

We asked the authors of this method to share the source code, so we could test it and benchmark it. Unfortunately, we did not receive any response so far.

2.5 Distributional Learning of Some Context-Free Languages with a Minimally Adequate Teacher

Angluin showed that the class of regular languages can be learned from a Minimally Adequate Teacher (MAT) using membership and equivalence queries [17]. Clark et al. showed that there is a class of context-free languages, that includes the class of regular languages, that can be learned in polynomial time from a MAT [18].

Clark proposes an inference algorithm for a class of context-free languages using a MAT [19], that is an extension of Angluin's *LStar* algorithm. The MAT can answer two types of queries: membership queries - for a given string, the learner can find out if that string is or is not in the language; and equivalence queries — the learner provides a hypothesis, and the teacher will either confirm that it is correct, or it will provide a counter example if it is not.

2.5.1 Assumptions

- we have a large number of correct programs
- we have a Minimally Adequate Teacher (MAT) that can answer membership and equivalence queries
- the language to be learned belongs to the set of all languages definable by a congruential CFG (Definition 4.5), which includes all regular languages (have only a finite number of congruence classes) and all NTS languages (the language defined by a CFG with the property: the set of sentential forms the CFG generates is unchanged when the rules are used both ways [20])
- not all context-free languages can be learned by using this approach (in particular, languages which are a union of infinitely many congruence classes are not).
- the production rules are in Chomsky Normal Form (this possibly limits the class of languages that can be represented, since the classic algorithm to convert any given context-free grammar to CNF [21] is not proven in [19], to guarantee to preserve the class of congruent context-free grammars)

2.5.2 Algorithm

The algorithm starts by building an Observation Table (which contains a set of strings K, a set of contexts F, and a set of grammatical strings D). After initializing the Observation Table, it adds data to it using the membership oracle, provided by the MAT. If the table is consistent, a CFG is built from it. The MAT is used again to check if the language defined by the newly created CFG is equivalent with the language to be learned.

If the MAT answers positively, the algorithm stops; otherwise, MAT provides a counter-example. If the MAT discovers overgeneration (the newly created CFG generates strings that are not in the language), the algorithm adds more features to the observation table. If, on the other hand, the MAT discovers undergeneration (the newly created CFG cannot generate all strings in the language), the algorithm adds more strings to K, to increase the number of equivalence classes.

This algorithm queries the MAT and adds information to the Observation Table until the MAT positively answers the membership query.

2.5.3 Implementation

Clark provides a sample run of the algorithm on the Dyck language (language consisting of balanced strings of parentheses) over the alphabet $\Sigma = \{a, b\}$. He also explains what happens with the Observation Table step by step and how the CFG looks after each iteration.

In the original paper, there is no evidence of testing the algorithm with bigger samples. We implemented this algorithm and tested it with multiple samples. The algorithm description, implementation details and experiment results can be found in Section 4.3.

2.6 A Polynomial Algorithm for the Inference of Context Free Languages

Clark et al. propose a polynomial algorithm for the inference of languages that have two special characteristics: the finite context property and the finite kernel property [22]. These characteristics are hold by all regular languages, many context-free languages, and some context-sensitive languages. This class of languages does not include all CFLs, since not all CFLs have those two characteristics. However it does include languages like the Dyck languages (language consisting of balanced strings of parentheses) of arbitrary order.

The authors of this paper present a rich grammatical formalism, called Binary Feature Grammars (BFG). The class of languages defined by BFGs contains all context-free languages (CFL) and some non context-free languages. They propose an algorithm that can identify in the limit, from positive data and a membership oracle, the class of CFLs with two properties: finite context and finite kernel. Their approach is based on a generalisation of distributional learning, following the work of [18].

2.6.1 Assumptions

- we have a set of positive data
- we have a membership oracle (an oracle that can tell the learner if a string is in the language or not)
- the language to be learned has finite context [22] (every string has a finite fiducial feature set)
- the language to be learned has finite kernel [22] (a finite set $K \subseteq \Sigma^*$ is a kernel for a language L, if for any set of features F, $L(G_0(K, F, L)) \supseteq L$)

2.6.2 Algorithm

The algorithm starts by initializing K, D, F, where K is a finite set of strings, F is a set of contexts, and D is a list of strings that have been seen so far. After initializing this structures, the algorithm examines the set of strings $T = Con(D) \odot Sub(D)$, where $Con(D)$ is a set of all contexts of a set of strings, and $Sub(D)$ is a set of non-empty substrings for a set of strings.

If the current hypothesis generates some element of this set that is not in the language, then it is overgeneralising: we need to add features. If on the other hand we undergeneralise, then we add all of the substrings of D to K, and all possible context to F. Using a membership oracle, a CFG is built from the information stored in K, D and F.

Clark et al. proved that this algorithm runs in polynomial time in the size of the sample, and makes a polynomial number of calls to the membership oracle [22]. They also proved that the same algorithm identifies in the limit the class of CFLs with the finite context property and the finite kernel property.

2.6.3 Implementation

The main contribution of this paper is to show that efficient learning is possible, with an appropriate representation [22]. The method presented in this paper rely on using a membership oracle, but it can be altered to get a PAC-learning result (Probably Approximately Correct).

Other than multiple formal theorems and proofs, Clark et al. did not present any information about the implementation of this algorithm or any experimental results. We asked the authors of this method to share the source code, so we could test it and benchmark it. One of the authors answered saying that he will share the code after he tidies it up. We asked again, but we did not receive the source code so far.

2.7 Exbar: Faster Algorithms for Finding Minimal Consistent DFAs

Lang describes **Exbar**, a powerful algorithm for the exact inference of minimal deterministic automata from given training data (positive and negative) [23]. This algorithm achieves the highest performance at that time (1999), on a set of graded benchmark problems that has been posted by Arlindo Oliveira. Lang compares Exbar with the inexact program **ed-beam**, which resulted from the Abbadingo DFA learning competition.

Before Exbar, an algorithm known as **bica** was considered to be state-of-art in the grammatical inference community. Bica was introduced by Oliveira one year before Exbar was developed [24]. From the results showed by Lang, Exbar can solve 774 problems (from a total of 810 benchmark problems) in under 1.1 seconds each, which is four orders of magnitude faster than bica.

2.7.1 Assumptions

- we have a set of positive and negative training data (marked as such)
- the output of this algorithm is a minimal DFA

2.7.2 Algorithm

The algorithm starts with building the Augmented Prefix Tree Acceptor (Definition 3.6), which embodies the entire training set (positive and negative examples). Then, the APTA is folded up into a smaller hypothesis by merging various pairs of compatible nodes. The algorithm operates in the red-blue framework and considers only merges between red and blue nodes. The basic operation of an algorithm in this framework is to select a blue node, and either accept it by recoloring it to red, or dispose of it by merging it with a red node.

The order in which nodes are picked and merged influence the overall performance of the inference algorithm. One improvement that Exbar has over older algorithms is that the former avoids latent conflicts by immediately trying to get rid of a blue node. The criteria used by Exbar, in picking a blue node is firstly choosing the blue nodes that can be disposed in the fewest ways. The best kind of blue node, according to this criteria, is the node that cannot be merged with any red node, which ends up with promoting that node to red. The next best kind is the node that has only one possible merge and so forth.

When considering merging between a blue node and a red node, one must first check if this merge respects the "determinization" rule (the entire tree rooted at the blue node can be merged with the graph rooted at the red node - Definition 3.7).

2.7.3 Implementation

In order to measure its performance, Exbar was tested on 810 benchmarking problems (each problem containing 20 training strings of length 30). In the experiment results, Lang also compares Exbar with bica, showing that the former has better performance for the benchmarking problems proposed by Arlindo Oliveira.

In the original paper, there is no evidence of testing the algorithm with bigger samples. We implemented this algorithm and tested it with multiple samples. The algorithm description, implementation details and experiment results can be found in Section 3.3.

2.8 EDSM: Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm

Abbadingo One DFA Learning Competition was designed to encourage work on algorithms that scale well both to larger DFAs and to sparser training data. The task was DFA learning from training data, consisting of both positive and negative examples. As a result of this competition, two state-of-art algorithms for the exact inference of minimal DFA were proposed.

Lang et al. describe one of these algorithms, the winning algorithm of Rodney Price (called EDSM), which orders state merges according to the amount of evidence in their favor [25]. EDSM is a simple and effective method for DFA induction from positive and negative examples. During the competition, Rodney Price realized that in the average case, a possible merge that calculates more label comparison is more likely to be valid, because each comparison verifies and strengthens the hypothesis or invalidates it immediately.

2.8.1 Assumptions

- we have a set of positive and negative training data (marked as such)
- the output of this algorithm is a minimal DFA

2.8.2 Algorithm

Similar to Exbar (Section 3.3), the algorithm starts with building the Augmented Prefix Tree Acceptor (APTA - Definition 3.6), which embodies the entire training set (positive and negative examples). After that, the APTA is folded up into a smaller hypothesis by merging various pairs of compatible nodes. The main insight of EDSM is avoiding bad merges (which cannot be directly detected when the training data is very sparse). This is done by computing merge scores, and firstly perform high scoring merges that have passed many tests, and therefore are likely to be correct.

Like other similar DFA inference algorithms that work with the APTA, the order of merges is essential in the overall performance of the method. If Exbar tries to perform merges as quick as possible, for EDSM is the opposite. EDSM calculates merge scores for all red-blue pairs and if any merge is valid, performs the merge with the highest score, or otherwise halt. A merge score is built as follows: for invalid merges (due to the conflicting labels), is $-\infty$; if there are no labels, is 0; otherwise, the number of labels minus one.

Because EDSM performs the merge scores for all possible red-blue pairs, it is important to reduce the number of these pairs. For that reason, EDSM operates in a framework slightly different from the well known red-blue framework. In this framework, the APTA is initially colored as follows: the root node is red, its first children are blue and any other node is white. As an observation, in the classical red-blue framework, we don't have white nodes, those nodes are blue.

2.8.3 Implementation

In order to measure its performance, EDSM was tested on 1000 random problems (each problem having 2000 training strings of length 0-15, and a depth -10 target DFA with about 64 states).

In the original paper, there is no evidence of testing the algorithm with bigger samples. We implemented this algorithm and tested it with multiple samples. The algorithm description, implementation details and experiment results can be found in Section 3.4.

2.9 The Use of Grammatical Inference for Designing Programming Languages

In the process of designing a new language (or extending an existing one), the designer needs a "natural" grammar for that language. The process can be split in three main steps: the designer prepares a set of sentences suitable for expressing the functions required of the language; the meaning for each sentence is made explicit by the designer; the order of the elementary steps defines a natural parsing order for the sentence.

Crespi-Reghizzi et al. consider the third step, and propose an interactive approach for designing a grammar, where the designer presents a sample of sentences and structures as input to a grammatical inference algorithm [26]. The algorithm builds a grammar which is a generalization of the examples submitted by the designer.

2.9.1 Assumptions

- we have a set of valid programs
- we have a set of pairs $(y_\mathrm{i}, m_\mathrm{i})$, where y_i is a sentence, and m_i is the corresponding interpretation sequence
- the language can be defined by a subclass of operator precedence grammars (to overcome this limitation, a more refined algorithm has been proposed by Crespi-Reghizzi in [27])
- we had defined the new constructs and we can exemplify them

2.9.2 Algorithm

Crespi-Reghizzi et al. suggest a method that heavily relies on the language designer, to help the algorithm converge on the target language, by asking positive and negative examples. Every time the learning algorithm conjectures a new grammar, it outputs all sentences for that grammar up to a certain length [3].

The designer types on a console a sample S_t of sentences (and structure descriptions) that he wants in the language. The algorithm infers a grammar G_t compatible with the sample, or otherwise outputs an error message. If the conjectured grammar is too large, there will be sentences in the output that don't belong and the designer marks them as such. If the conjectured grammar is too small, there will be sentences missing from the output and the designer is expected to provide them. The designer's corrections are fed back into the algorithm which corrects the grammar and outputs a new conjecture, and the process repeats until the target grammar is obtained [3].

2.9.3 Implementation

Crespi-Reghizzi et al. suggest (in the conclusion), that the algorithm is implemented and tested on examples of low complexity. However, the results are not shown [26]. Because this method heavily relies on the language designer, the results of one experiment cannot guarantee if the method is efficient or not.

Chapter 3

Deterministic Finite Automata

3.1 Main Aspects

As expressed in the Chomsky hierarchy (Figure 1.1), type-3 grammars (regular grammars) generate regular languages. These languages can be generated by a finite state automaton (also known as finite state machine). A deterministic finite automaton (DFA) is a finite state machine that accepts/rejects finite strings of symbols and only produces an unique computation of the automaton for each string [28].

Multiple different grammars can define the same language. In the same way, there can exist multiple different DFAs that accept the same language. When inferring a DFA from examples, it is desirable to find the smallest (minimal) DFA that accepts the target language, because the minimal DFA ensures minimal computational cost. Gold showed that finding the minimal DFA that accepts an unknown regular language from a finite set of positive and negative samples is NP-complete [29]. This means that the problem cannot be solved quickly (in polynomial time).

Several algorithms for inferring the minimal DFA were developed throughout the years. These algorithms generally start by building an augmented prefix tree acceptor (APTA) from positive and negative sample sets (also known as training sets), then perform merges in a particular order and by specific rules until no valid merge can be found. These merges are done in order to generalize the language accepted by the DFA.

In this chapter we analyze, implement and benchmark two algorithms, both considered state of the art for this task. These algorithms are Lang's Exbar [23] and Rodney Price's EDSM [25] (one of the winners of Abbadingo Learning Competition).

3.2 General Definitions

The general problem of grammatical inference is defined as follows:

Definition 3.1. Given sets of labeled example strings S_+ and S_- such that $S_+ \subset L(G)$ and $S_- \subset L'(G)$ infer a DFA (A) such that the language of A denoted $L(A) = L(G)$. $L(G)$ is a language generated from an unknown Type-3 grammar (G). Its complement, $L'(G)$, is defined as $L'(G) = \Sigma^* - L(G)$ where Σ^* is the set of all strings over the alphabet (Σ) of $L(G)$ [30].

Following Harrison [31], a formal definition of a **Type-3 grammar** (for regular languages) is given below:

Definition 3.2. $G = \{V, \Sigma, P, S\}$, where

- V is a finite set of variables / nonterminals,
- Σ is a finite set of terminals,
- $P \subseteq V \times (\Sigma \cup V)^*$ is a finite set of production rules,
- $S \in V$ is referred to as the start state / nonterminal,

and the rules in P are of the form

- $A \to aB$, or
- $A \to a$

where $A, B \in V$ and $a \in \Sigma$.

The **Deterministic Finite Automaton** (DFA) representing Type-3 grammars is a quintuple. Following Hopcroft et al. [32], a formal definition of a DFA(A) is given below:

Definition 3.3. $A = \{Q, \Sigma, \delta, s, F\}$ where

- Q is a finite non-empty set of states,
- Σ is a finite non-empty set of input symbols or input alphabet,
- $\delta : Q \times \Sigma \to Q$ the transition function,
- $s \in Q$ the start state,
- $F \subseteq Q$ the final states or accepting states of A.

Definition 3.4. A **training set** is a set of pairs $T = \{(s_1, l_1), ..., (s_m, l_m)\}$ where each pair $((s_i, l_i)) \in \Sigma^* \times \{0, 1\}$ represents one input string and the label (or output) that corresponds to that string [33].

Examples of training sets can be found in next sections (e.g. Subsection 3.6.1).

Definition 3.5. The **destination state** that results from the application of a string $s = (a_1, ..., a_k)$ is applied, denoted by $\delta(q, s)$, represents the final state reached by an automaton after a sequence of inputs $(a_1, ..., a_k)$, is applied in state q. This state is defined to be $\delta(q, s) = \delta(\delta(...\delta(\delta(q, a_1), a2), ...), a_k)$ [33].

The **augmented prefix tree acceptor** (APTA) is a 6-tuple. Following Coste et al. [34], a formal definition of an $APTA(S_+, S_-)$ is given below:

Definition 3.6. $A = \{Q, \Sigma, \delta, s, F_+, F_-\}$ where

- Q is a finite non-empty set of nodes,
- Σ is a finite non-empty set of input symbols or input alphabet,
- $\delta : Q \times \Sigma \rightarrow Q$ the transition function,
- $s \in Q$ the start or root node,
- $F_+ \subseteq Q$ identifying final nodes of strings in S_+,
- $F_- \subseteq Q$ identifying final nodes of strings in S_-.

Examples of APTA representations can be found in next sections (e.g. Figure 3.1b).

Definition 3.7. Two nodes q_i and q_j in the APTA are considered <u>not</u> **equivalent** if and only if:

1. $(q_i \in F_+$ and $q_j \in F_-)$ or $(q_i \in F_-$ and $q_j \in F_+)$, or
2. $\exists s \in \Sigma$ such than if $(q_i, s, q_{i'}), (q_j, s, q_{j'}) \in \delta$ then $q_{i'}$ not equivalent to $q_{j'}$

(implied in Moore's minimization algorithm [32])

3.3 Exbar

In 1999, Lang proposed a powerful new algorithm for the exact inference of minimal deterministic automata from given training data [23]. This algorithm is called **Exbar** and it achieves the highest performance (at that time) on a suite of benchmark problems. Before Exbar, an algorithm known as **bica** was considered to be state-of-art in the grammatical inference community. Bica was introduced by Oliveira one year before Exbar was developed [24].

3.3.1 Algorithm steps

1. Build the augmented prefix tree acceptor (APTA - Definition 3.6) from positive and negative example strings
   - APTA is built from beginning in the red-blue framework (the root node is red and all other nodes are blue)
   - for every sample string, construct a path (each character from that string is a labeled edge)
   - the last node of the new path is labeled (accepted or rejected, depending on the sample label)
   - e.g. for training set 1 (Subsection 3.6.1), the resulting APTA can be seen in Figure 3.1b
2. Generalize the accepted language by merging pairs of compatible nodes (Exbar)
   (a) pick blue node (look first for the blue nodes that cannot be merged with any of the red nodes and after that, look for blue nodes that have only one possible merge, two possible merges and so on)
   (b) try to merge (the merge is done applying the "determinization" rule (Definition 3.7)) or promote the blue node to red
   (c) start again until no valid merges can be found
3. Build the DFA from the current version of APTA (after all merges)

3.3.2 Algorithm description

Exbar algorithm starts with building the APTA, from there it considers candidate nodes that are adjacent to a core graph of accepted nodes, and for each candidate it decides whether to accept it or to view it as identical to a previously accepted node [23].

The order in which nodes are picked and merged matters, so the main difference between Exbar and previous algorithms is that the former avoids latent conflicts by immediately trying to perform additional merges. Some of the older algorithms compute merging scores and choose a candidate based on that score.

The criteria used by Exbar, in picking a candidate blue node, has a strong influence on the program's performance. It first chooses the blue nodes that can be disposed of in the fewest ways. The best kind of blue node is the node that cannot be merged with any red node, which ends up with promoting that node to red. The next best kind is the node that has only one possible merge and so forth.

3.3.3 Implementation and benchmarking

The Exbar algorithm is implemented based on the pseudo-code and description found in Lang's paper [23]. Due to performance concerns, the chosen programming language is **C++**, version 11 and Microsoft **Visual C++ 2013 compiler**. The C++ source code is available at https://bitbucket.org/orosu/thesis-cpp, and contains approximately 1400 lines of code in 7 classes.

Prior to this, we also implemented Exbar algorithm in Rascal but we did not continue with this language due to performance concerns. The Rascal source code is available at https://orosu@bitbucket.org/orosu/thesis-rascal, and contains approximately 400 lines of code in 5 modules.

In order to validate the correctness of the algorithm, we manually checked the results for the first three training sets (Subsections 3.6.1, 3.6.2 and 3.6.3). Figure 3.1 displays three states of APTA, built from training set 1 (Subsection 3.6.1).

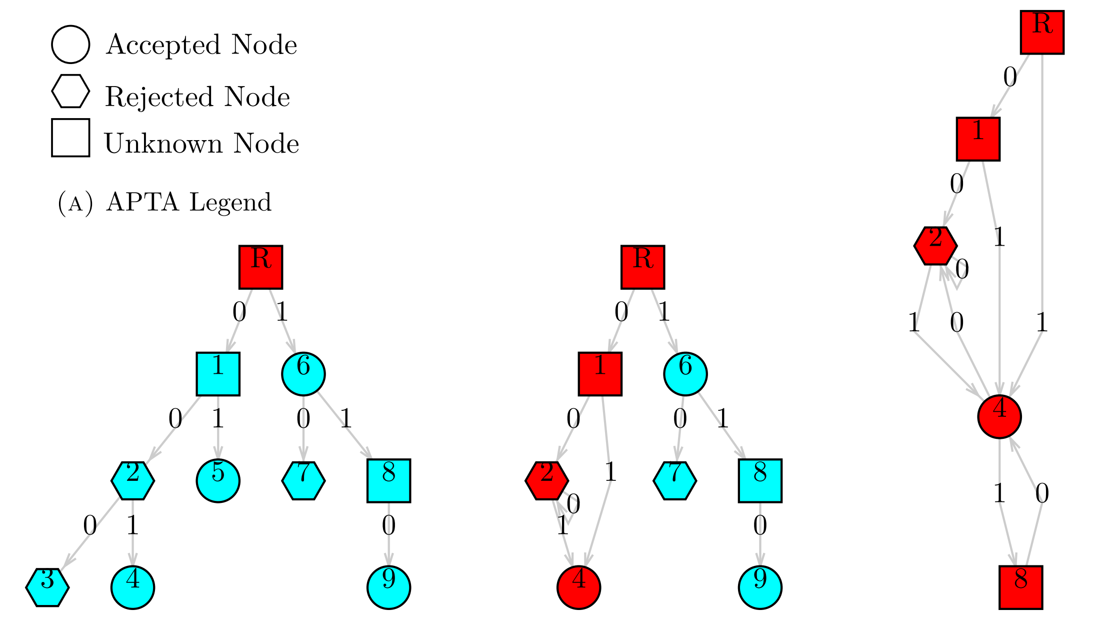
The program is compiled and run for all training sets (Section 3.6). The results can be seen in Table 3.1.

Sample	Training Set (clock_t)	APTA (clock_t)	Exbar (clock_t)
Training Set 3.6.1	1	57	581
Training Set 3.6.2	0	52	361
Training Set 3.6.3	0	52	365
Training Set 3.6.4	55	1315	12566
Training Set 3.6.5	49	1443	9235
Training Set 3.6.6	37	2402	x*

Note 1: clock_t = clock ticks (processor time consumed by the program)

Note 2: * not successful (stack overflow)

TABLE 3.1: Benchmarking results Exbar



(A) APTA Legend

(B) APTA state 0 (before any merge)

(C) APTA state i

(D) APTA state n

FIGURE 3.1: APTA states for Training Set 1 - Exbar (generated from C++)

3.4 Evidence Driven State Merging

During the Abbadingo One DFA Learning Competition, one of winners, Rodney Price, realized that in the average case, a possible merge that calculates more label comparison is more likely to be valid, because each comparison verifies and strengthens the hypothesis or invalidates it immediately. This observation led to the development of evidence driven state merging algorithm, also known as EDSM [25]. This algorithm performs merges in a particular order according to the amount of evidence it has about them.

3.4.1 Algorithm steps

1. Build the augmented prefix tree acceptor (APTA - Definition 3.6) from positive and negative example strings
   - similar to building the APTA for Exbar algorithm, with one small difference (the root node is red, its first level children are blue and the rest of the nodes are white)
   - e.g. for training set 1 (Subsection 3.6.1), the resulting APTA can be seen in Figure 3.2b
2. Generalize the accepted language by merging pairs of compatible nodes (EDSM)
   (a) evaluate all possible merges between red and blue nodes and compute a score for every merge (conflicting labels: $-\infty$, no labels: 0, otherwise: number of labels - 1)
   (b) if there are any blue nodes that cannot be merged with any red nodes, promote them and start over
   (c) choose the merge with the highest score
   (d) the merging is done similar to the merging from Exbar algorithm, with one difference, the former also needs to take care of coloring the white nodes into blue nodes
   (e) start again until no valid merges can be found
3. Build the DFA from the current version of APTA (after all merges)

3.4.2 Algorithm description

Similar to Exbar, EDSM starts with building the APTA, from there it evaluates all possible merges between red and blue nodes in order to compute merge scores. A merge score is built as follows: for invalid merges (due to the conflicting labels), is $-\infty$; if there are no labels, is 0; otherwise, the number of labels minus one [25].

The main difference between Exbar and EDSM is that the latter avoids bad merges by doing high score merges first. These merges already passed many tests and have a higher probability to be valid.

Another difference that needs to be considered is that EDSM also has to color impacted white nodes to blue, when one blue node is promoted to red or when a blue node is merged with a red node.

3.4.3 Implementation and benchmarking

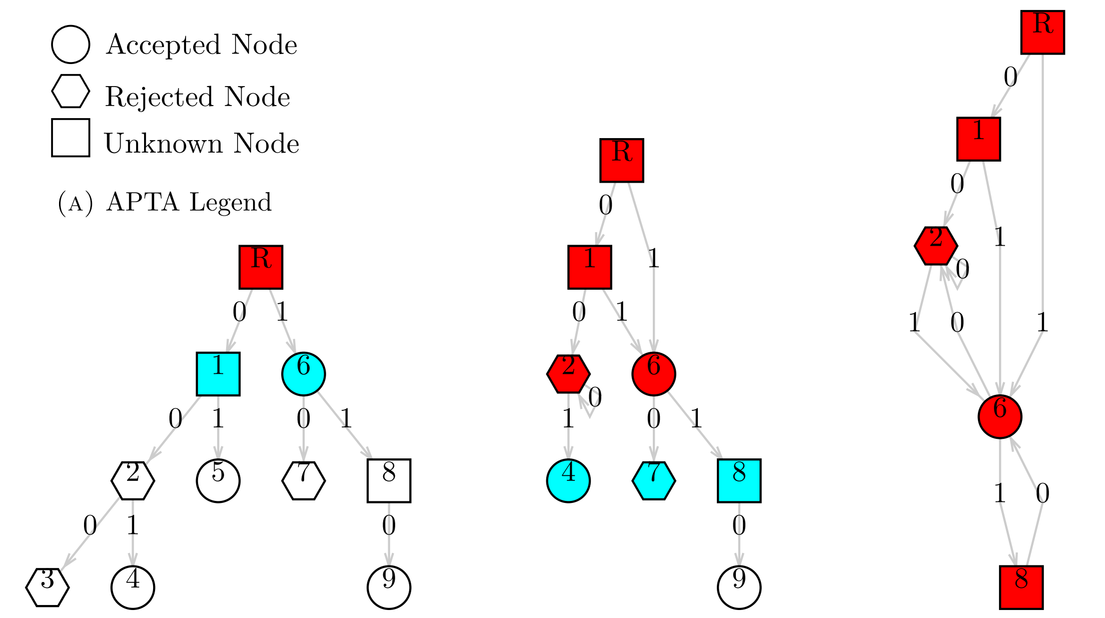
The EDSM algorithm is implemented based on the pseudo-code and description found in Lang's paper [25] and also in the book "Grammatical Inference: Learning Automata and Grammars" [35]. Due to performance concerns, the chosen programming language is **C++**, version 11 and Microsoft **Visual C++ 2013 compiler**. The C++ source code is available at https://bitbucket.org/orosu/thesis-cpp, and contains approximately 1500 lines of code (there are a few reused classes from Exbar) in 7 classes.

Prior to this, we also implemented EDSM algorithm in Rascal but we did not continue with this language due to performance concerns. The Rascal source code is available at https://orosu@bitbucket.org/orosu/thesis-rascal, and contains approximately 400 lines of code (there are a few reused modules from Exbar) in 5 modules.

In order to validate the correctness of the algorithm, we manually checked the results for first three training sets (Subsections 3.6.1, 3.6.2 and 3.6.3). Figure 3.2 displays three states of APTA, built from training set 1 (Subsection 3.6.1).

Although the merges are done in different order, due to different search algorithms, we can notice that the last APTA visualization, representing the final state, is almost the same with the corresponding one from Exbar. We can also notice that at state **i**, the APTA visualizations are quite different, again due to the order of merges.

The program is compiled and run for all training sets (Section 3.6). The results can be seen in Table 3.2.



(A) APTA Legend

(B) APTA state 0 (before any merge)

(C) APTA state i

(D) APTA state n

FIGURE 3.2: APTA states for Training Set 1 - EDSM (generated from C++)

Sample	Training Set (clock_t)	APTA (clock_t)	EDSM (clock_t)
Training Set 3.6.1	1	99	633
Training Set 3.6.2	0	54	372
Training Set 3.6.3	0	58	372
Training Set 3.6.4	55	1687	239776
Training Set 3.6.5	49	1886	323615
Training Set 3.6.6	37	2400	x*

Note 1: clock_t = clock ticks (processor time consumed by the program)

Note 2: * not successful (>10 minutes)

TABLE 3.2: Benchmarking results EDSM

Insights from replication

While benchmarking EDSM, we noticed that in its original form, the algorithm was computing merge scores for all possible red - blue combinations before every search. Because of that, it was not practical for larger training sets (e.g. the training sets from files). In order to mitigate this problem, we augmented the algorithm to calculate all merge scores only one time, and after each merge or promotion, compute the merge scores between the nodes effected by that action. In this way, the execution time decreased and we were able to test with first two training sets from file (3.6.4 and 3.6.5). The modification changed the merge order but the end result is equivalent with the original.

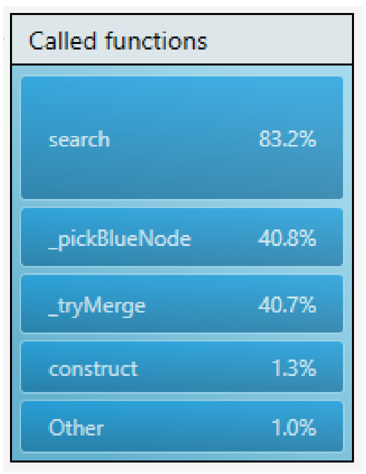
3.5 Discussion

In previous two sections we analysed, implemented and benchmarked two state of the art algorithms for inferring minimal consistent DFAs. These algorithms are Exbar (Section 3.3) and EDSM (Section 3.4). We compiled and tested these algorithms in the same way, using the same training sets. The results for each test can be found in the previous sections.

Both of the algorithms work in similar ways. They start by building the APTA and then they merge nodes until all valid merges are exhausted. The order in which the nodes are merged differs. This difference is quite obvious in the running time, where Exbar algorithm outperforms EDSM algorithm for all training sets. Exbar never creates latent conflicts because it immediately tries to perform the merge, while EDSM computes all merge scores and after that it performs the merge with the highest score.

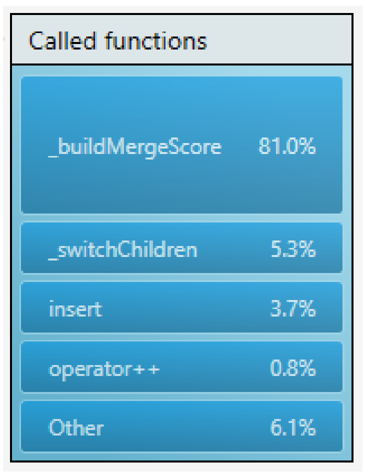
In the original paper [23], Exbar was tested on 810 benchmark problems (each problem containing 20 training strings of length 30). In a similar way, in the original paper [25], EDSM was tested on 1000 random problems (each problem having 2000 training strings of length 0-15, and a depth -10 target DFA with about 64 states). The results of both tests can be seen in the original papers. We did not go that far because performance considerations made it clear that the method is unacceptable for us.

In order to identify the hot paths, we profiled both algorithms individually, using Training Set from file 1 (Subsection 3.6.4). The results of this profiling can be seen in Figure 3.3.



Called functions	
search	83.2%
_pickBlueNode	40.8%
_tryMerge	40.7%
construct	1.3%
Other	1.0%

(A) Profiling Exbar



Called functions	
_buildMergeScore	81.0%
_switchChildren	5.3%
insert	3.7%
operator++	0.8%
Other	6.1%

(B) Profiling EDSM

Figure 3.3: Profiling Exbar and EDSM

Analysing the data gathered through profiling the algorithms, we noticed that while Exbar spends ~83% of the running time searching and merging (half on picking the blue node and half on merging), EDSM spends ~81% of the running time building the merge scores. This highlights the fact that EDSM's search for the highest score is too expensive to be practically used for large training sets.

Although Exbar algorithm proved higher performance, it is still not suitable for inferring a grammar from source code. The reason is that it analyses training sets, char by char. For a new string, it builds a tree path of length equal to the number of characters in the string. For large training sets, these trees become too big to be efficiently handled.

3.6 Training sets

3.6.1 Training set 1

T_1 = `{<"1", true>, <"110", true>, <"01", true>, <"001", true>, <"00", false>, <"10", false>, <"000", false>}`

3.6.2 Training set 2

T_2 = `{<"1", true>, <"11", true>, <"1111", true>, <"0", false>, <"101", false>}`

3.6.3 Training set 3

T_3 = `{<"a", true>, <"abaa", true>, <"bb", true>, <"abb", false>, <"b", false>}`

3.6.4 Training set from file 1

TF_1 = random file that contains php source code

size(TF_1) = **7.76 KB** (7,949 bytes)

3.6.5 Training set from file 2

TF_2 = random file that contains php source code

size(TF_2) = **11.8 KB** (12,130 bytes)

3.6.6 Training set from file 3

TF_3 = random file that contains php source code

size(TF_3) = **17.9 KB** (18,406 bytes)

Chapter 4

Context-Free Grammars

4.1 Main Aspects

As expressed in the Chomsky hierarchy (Figure 1.1), type-2 grammars (context-free grammars) generate context-free languages (CFLs). A context-free grammar (CFG) is a formal grammar in which every production rule can be applied regardless of the context of a nonterminal. These grammars generate the next smallest class of languages, the context-free languages, which include all regular languages.

Similar to regular grammars, multiple different CFGs can generate the same CFL. Polynomial-time algorithms to learn this class of grammars, have also been investigated. Identifying context-free grammars in polynomial time is considerably more difficult than for DFAs, so most polynomial results in the literature either learn a strict subset of CFGs, use structured strings as input, or both [3]. Unlike DFA inference, there is currently no known algorithm to identify a general CFL from positive and negative samples in polynomial time.

In 2008, Clark et al. proposed a polynomial algorithm for the inductive inference of languages that exhibit two constraints on the context distributions [22]. The algorithm is based on positive data and a membership oracle. Two years later, Clark proposed another algorithm for learning CFLs from a Minimally Adequate teacher providing membership and equivalence queries [19].

In this chapter we analyze, implement and benchmark the algorithm proposed by Clark for learning some context-free languages with a Minimally Adequate Teacher [19].

4.2 General Definitions

Notations (according to [22] and [19]):

- Σ is a finite non-empty **alphabet**, which is known
- Σ^* is the set of all **finite strings** of Σ
- λ is the **empty string**
- Language L is a **subset** of Σ^*
- uv is the **concatenation** of u and v
- $u \in \Sigma^*$ is a **substring** of $v \in \Sigma^*$, if there are strings $l, r \in \Sigma^*$ such that $v = lur$
- a **context** (l, r) is a pair of strings, an element of $\Sigma^* \times \Sigma^*$
- the **distribution of a string** is $C_{\mathrm{L}}(w) = \{(l, r) | lwr \in L\}$ (set of all contexts of a string)
- the **distribution of a set of strings** is $C_{\mathrm{L}}(S) = \cup_{w \,\in\, S} C_{\mathrm{L}}(w)$
- for a string u and a context $f = (l, r)$, $f \odot u = lur$ is the **insertion** or wrapping operation (combines a context with a substring)
- $Sub(w) = \{u | \exists l, r \in \Sigma^*, lur = w\}$ is the set of all **substrings of a string** w
- $Sub(S) = \cup_{w \,\in\, S} Sub(w)$ is the set of all **substrings of a set of strings** S
- $u \equiv_{\mathrm{L}} v$ means two strings are **congruent** with respect to a language L, if and only if $C_{\mathrm{L}}(u) = C_{\mathrm{L}}(v)$ (**equivalence relation**)
- $[u]_{\mathrm{L}} = \{v | u \equiv_{\mathrm{L}} v\}$ is the **equivalence class** of u

Lemma 4.1. *For any language L, if $u \equiv_L u'$ and $v \equiv_L v'$ then $uv \equiv_L u'v'$ [19].*

This means that the **concatenation between two classes** is $[u] \circ [v] = [uv]$. This is a monoid, as it is associative and contains a unit $[\lambda]$; it is called the syntactic monoid: $\Sigma^* / \equiv_{\mathrm{L}}$.

Definition 4.2. A **context-free grammar** (CFG) is a quadruple $G = (\Sigma, V, P, S)$. Σ is a finite alphabet of terminal symbols, V is a set of non terminals such that $\Sigma \cap V = \emptyset, P \subseteq V \times (V \cup \Sigma)^*$ is a finite set of productions, $S \in V$ is the start symbol [22].

Having a production of $P : N \rightarrow \alpha$ with $N \in V$ and $\alpha \in (V \cup \Sigma)^+$, we write $uNv \Rightarrow_{\mathrm{G}} u\alpha v$ if there is a production $N \rightarrow \alpha$ in G. $\overset{*}{\Rightarrow}_{\mathrm{G}}$ denotes the **reflexive transitive closure** of $\Rightarrow_{\mathrm{G}}$ [22].

Definition 4.3. An **objective context-free grammar** is a quadruple $G = (\Sigma, V, P, I)$. Σ is a finite alphabet of terminal symbols, V is a set of non terminals such that $\Sigma \cap V = \emptyset$, $P \subseteq V \times (V \cup \Sigma)^+$ is a finite set of productions, I is a non-empty subset of V, the set of initial symbols [19].

Note 1: There are multiple start symbols.

Note 2: We will consider only the case of CFGs in Chomsky Normal Form(CNF), which means that all productions are of the form $N \rightarrow PQ$, or $N \rightarrow a$ or $N \rightarrow \lambda$.

Note 3: One example of a CFG representation can be seen in Figure 4.1.

Definition 4.4. The language defined by a CFG G is $L(G) = \{w \in \Sigma^* | S \overset{*}{\Rightarrow}_{\mathrm{G}} w\}$ [22].

Note 1: One example of a CFL is the Dyck language $L = \{a^n b^n | n \geq 0\}$.

Definition 4.5. A **context-free grammar** (CFG) G is **congruential** if for every nonterminal N it is the case that $L(G, N)$ is a subset of a congruence class of $L(G)$; that is if $N \overset{*}{\Rightarrow} u$ and $N \overset{*}{\Rightarrow} v$ implies that $u \equiv_{\mathrm{L(G)}} v$ [19].

Note 1: We can assume that we only have one nonterminal for each congruence class. If we have two nonterminals that generate strings from the same congruence class, then we can merge them.

Definition 4.6. $\mathrm{L}_{\mathrm{CCFG}}$ is the set of all languages definable by a congruential CFG (CCFG) [19].

Note 1: $\mathrm{L}_{\mathrm{CCFG}}$ includes all regular languages.

Note 2: Not all CFLs are in $\mathrm{L}_{\mathrm{CCFG}}$.

Definition 4.7. An **observation table** consists of a non-empty finite set of strings K, a non-empty finite set of contexts F and a finite function mapping $F \odot KK$ to $\{0, 1\}$. Given a context (l, r) in F, and a substring $W \in KK$, we have a 1 (one) in the table if lwr is in the language and a 0 (zero) if it is not. We will represent this observation table as a tuple $\langle K, D, F \rangle$, where D is the set of grammatical strings in $F \odot KK$ [19].

Note 1: One example of an observation table representation can be seen in Table 4.2.

Definition 4.8. For two strings $u, v \in KK$, we say they are **equivalent** if they appear in the same set of contexts, and we write $u \sim_{\mathrm{F}} v$ [19].

Definition 4.9. An observation table $\langle K, D, F \rangle$ is **consistent** if for all $u_1, u_2, v_1, v_2 \in K$, if $u_1 \sim_{\mathrm{F}} u_2$ and $v_1 \sim_{\mathrm{F}} v_2$, then $u_1 u_2 \sim_{\mathrm{F}} v_1 v_2$ [19].

Note 1: If a table is not consistent, we know that we do not have a large enough set of features.

4.3 Learning with MAT

Angluin showed that regular languages can be learned from a Minimally Adequate Teacher (MAT) using membership and equivalence queries [17]. Using the same approach, in 2007, Clark et al. showed that some CFGs can be identified in the limit from positive data alone, by identifying the congruence classes of the language [18]. Three years later, Clark showed that there is a natural class of CFLs, which includes all regular languages, that can be learned from a MAT in polynomial time [19]. He proposed a new algorithm that is an extension of the classic and well-studied LSTAR (also known as L^*) algorithm, that was introduced by Angluin [17].

A MAT is assumed to answer correctly two types of questions (membership and equivalence) from the learner about the language to be learned. This teacher must be able to test a hypothesis and reply whether it is correct, and supply a counter-example if not. Clark proposed a stochastic setting [19], in which the learner gets an approximately correct hypothesis with high probability. In this scenario, the teacher can be replaced by a random sampling oracle.

4.3.1 Algorithm steps

4.3.1.1 Initialization

1. Setup the language to be learned L, from positive samples: read line by line from file(s) and build the alphabet Σ of the language while adding samples to L
    - e.g. for the Dyck language (language consisting of balanced strings of parentheses) over the alphabet $\Sigma = \{a, b\}$, the language $L = \{\lambda, ab, abab, aabb, ...\}$
2. Setup the Minimally Adequate Teacher (MAT) over the language L, which can answer membership and equivalence queries
    - membership - the learner can find out if a string is in the language L
    - equivalence - the learner provides the teacher with a CFG and the teacher will either confirm if the language defined by that grammar is equivalent with L, or it will provide a counter-example (it checks for both: overgeneration and undergeneration)
3. Build the Observation Table (Definition 4.7)
    - initialise $K = \{\lambda\}$, $F = \{(\lambda, \lambda)\}$ and $D = \{\lambda\}$
    - this will be the main data type used to build the CFG

4.3.1.2 Learn CFG

Execute the following steps until the equivalence query confirms that the language defined by the CFG is equivalent with L:

1. Make the context-free grammar from the current version of the observation table
   - build the table and populate D from K and F using the membership oracle
   - make the observation table consistent (Definition 4.9)
   - split the elements in K into equivalence classes, associate nonterminals to those classes, and build production rules from these equivalence classes
   - the grammar is congruential (Definition 4.5) and in Chomsky Normal Form (all of its production rules are of the form $N \to PQ$ or $N \to a$ or $N \to \lambda$)
2. Query the teacher to confirm that the newly made grammar is correct (equivalence)
   - check if the grammar undergenerates (there are strings in L that cannot be generated by this grammar)
   - check if the grammar overgenerates (there are strings that can be generated by the grammar, which are not in L)
3. If the grammar is correct, return it and terminate
4. If the grammar is incorrect, get the counter-example and continue
   - if the grammar undergenerates, add positive counter-example w: get all substrings of w and add them to K (to increase the number of congruence classes)
   - if the grammar overgenerates, add more contexts until the grammar no longer generates the counter-example
5. Go to step 1 and execute the same steps again, until the grammar is correct

4.3.2 Algorithm description

This algorithm starts with building an observation table (Definition 4.7) and continues with filling it by membership querying the MAT. If the table is consistent, a CFG is generated from it. Based on this grammar, the MAT is queried again to determine if the language defined by this grammar is equivalent with the language to be learned.

If the grammar is correct, the algorithm terminates; otherwise if the MAT discovers overgeneration (generates strings that are not in the language), it adds more features until the grammar no longer generates the string. If, on the other hand it discovers undergeneration (does not generate all strings in the language), it adds more strings to K to increase the number of congruence classes of the language defined by the CFG.

In order to test if all strings in L (language to be learned) are generated by the CFG, we implemented the CYK algorithm [36]. The purpose of this algorithm is to decide if a string is in a context-free language or not. It is a good match for our MAT because the production rules of the CFG are already in CNF.

4.3.3 Implementation and benchmarking

This algorithm is implemented based on the pseudo-code and description found in Clark's paper [19]. Due to performance concerns, the chosen programming language is **C++**, version 11 and Microsoft **Visual C++ 2013 compiler**. The C++ source code is available at https://bitbucket.org/orosu/thesis-cpp, and contains approximately 2000 lines of code in 12 classes.

In order to validate the correctness of the algorithm, we manually checked the results for the first three samples (Subsections 4.4.1, 4.4.2 and 4.4.3). Figure 4.1 displays the CFGs after each iteration of the algorithm for sample 1 (Subsection 4.4.1).

The program is compiled and run for all training sets (Section 4.4). The results can be seen in Table 4.1. We can notice that this algorithm is not suitable for long sample size (in our case, number of characters per line) because of the way Observation Table works. The problem is that the table becomes very big, very quick because it needs to contain all possible substrings of a sample string.

In order to identify the hot paths, we profiled the algorithm, using Training Set 3 (Subsection 4.4.3). Based on the results of this profiling, we identified the hottest execution path. This hot path is $ObservationTable :: MakeConsistent()$ with more that 98% of the execution resources. This method is called before making a CFG and it makes the observation table consistent. The reason why this method consumes so many resources is the heavy method itself. It looks for four strings that satisfy the following criteria: u_1, u_2, and v_1, v_2 in K, and $(l, r) \in F$ such that $u_1 \sim_{\mathrm{F}} u_2, v_1 \sim_{\mathrm{F}} v_2, lu_1v_1r \in D$, and $lu_2v_2r \notin D$.

If we compare this algorithm with the two implemented for DFA, we notice that the latter algorithms can actually parse source code (files with more than 12 KB), while the CFG algorithm described above can only practically parse short sample strings (less than 10 characters per sample). The output of these two types of algorithms is different. While Exbar and EDSM algorithms output minimal DFAs, the algorithm presented in this section outputs a CFG with terminals, nonterminals and production rules.

Sample	Number of iterations	Learning with MAT (clock_t)
Training Set 4.4.1	2	25
Training Set 4.4.2	4	118
Training Set 4.4.3	7	12492
Training Set 4.4.4	3	735500
Training Set 4.4.5	3	2737323
Training Set 4.4.6	x*	x*

Note 1: clock_t = clock ticks (processor time consumed by the program)

Note 2: * not successful (>120 minutes)

TABLE 4.1: Benchmarking results Learning with MAT

K\F	(λ, λ)
λ	1

(A) Step 0

K\F	(λ, λ)	(λ, b)
λ	1	0
a	0	1
ab	1	0
b	0	0

(B) Step 1 - each row is K

K\F	(λ, λ)	(λ, b)
aa	0	0
aab	0	1
aba	0	1
abab	1	0
abb	0	0
ba	0	0
bab	0	0
bb	0	0

(C) Step 1 - each row is $KK\setminus K$

TABLE 4.2: Learning with MAT - Sample 1 Observation Table

$$\begin{aligned} \Sigma &= \{\lambda\} \\ V &= \{S\} \\ I &= \{S\} \\ P &= \{S \to \ \lambda\} \end{aligned}$$

(A) Step 0

$$\begin{aligned} \Sigma &= \{\lambda, a, b\} \\ V &= \{A, B, S\} \\ I &= \{S\} \\ P = \{A \to \ AS,\ A \to \ SA,\ A \to \ a,& \\ B \to \ b,\ S \to \ \lambda,\ S \to \ AB,\ S \to \ SS\}& \end{aligned}$$

(B) Step 1

FIGURE 4.1: Learning with MAT - Sample 1 CFG

Insights from replication

While benchmarking, we noticed that in its original form, the algorithm was adding nonterminals for every equivalence class. Because this equivalence class can be empty, it can produce unneeded productions and therefore, unneeded nonterminals. In order to mitigate this problem, we augmented the algorithm to skip empty equivalence classes. For the three small samples, there were no noticeable performance drawbacks. We did notice that for sample 1 (Subsection 4.4.1), instead of three steps of iterations and nine production rules in the end, we now have only two steps of iterations and seven production rules. For the tested samples, this modification produced smaller grammars that define the same language.

K\F	(λ, λ)
λ	1

(A) Step 0

K\F	(λ, λ)	(λ, y)	(x, λ)	(x+y, λ)	(y, λ)
λ	1	1	1	1	1
+y	0	0	1	0	0
x	1	0	0	0	0
x+	0	1	0	0	0
x+y	1	0	0	0	0
y	1	0	0	0	0
+y+y	0	0	0	0	0
+yx	0	0	0	0	0
+yx+	0	0	0	0	0
+yx+y	0	0	0	0	0
+yy	0	0	0	0	0
x++y	0	0	0	0	0
x+x	0	0	0	0	0
x+x+	0	0	0	0	0
x+x+y	0	0	0	0	0
x+y+y	0	0	0	0	0
x+yx	0	0	0	0	0
x+yx+	0	0	0	0	0
x+yx+y	0	0	0	0	0
x+yy	0	0	0	0	0
xx	0	0	0	0	0
xx+	0	0	0	0	0
xx+y	0	0	0	0	0
xy	0	0	0	0	0
y+y	0	0	0	0	0
yx	0	0	0	0	0
yx+	0	0	0	0	0
yx+y	0	0	0	0	0
yy	0	0	0	0	0

(B) Step 3

TABLE 4.3: Learning with MAT - Sample 2 Observation Table

$$\begin{gathered}\Sigma = \{\lambda\}\\ V = \{S\}\\ I = \{S\}\\ P = \{S \to\ \lambda\}\end{gathered}$$

(A) Step 0

$$\begin{gathered}\Sigma = \{\lambda, x, y\}\\ V = \{S, S1\}\\ I = \{S, S1\}\\ P = \{S \to\ x,\ S \to\ y,\\ S1 \to\ \lambda\}\end{gathered}$$

(B) Step 2

$$\begin{gathered}\Sigma = \{\lambda, +, x, y\}\\ V = \{A, B, C, S, S1\}\\ I = \{S, S1\}\\ P = \{A \to\ SC,\ B \to\ CS,\\ C \to\ +,\ S \to\ SB,\ S \to\ x,\\ S \to\ y,\ S1 \to\ \lambda\}\end{gathered}$$

(C) Step 3

FIGURE 4.2: Learning with MAT - Sample 2 CFG

4.4 Samples

4.4.1 Sample 1

S_1 = `{"ab", "abab", "aabb", "aababb"}`

Note 1: The Dyck Language.

4.4.2 Sample 2

T_2 = `{"x", "y", "x+y"}`

4.4.3 Sample 3

T_3 = `{"x", "y", "x+y", "x*y", "x/y", "x-z", "x+y*x", "x/y-z"}`

4.4.4 Sample 4

T_4 = `a simple php file which outputs the empty string`

```
<?php
echo '';
```

4.4.5 Sample 5

T_5 = `a simple php file which outputs "hi"`

```
<?php
echo 'hi';
```

4.4.6 Sample 6

T_6 = `a simple php file which outputs "hello"`

```
<?php
echo 'hello';
```

Conclusion

In this thesis we addressed the problem of inferring a grammar from source code written in an unknown programming language. We started by defining basic notions of grammars and language learning, and we briefly described the characteristics of the main learning models used in grammatical inference. We studied and analysed existing grammatical inference methods for programming languages, and we addressed their limitations and applicability in the domain of programming languages.

We have chosen two methods for DFA learning and one for CFG learning, and we implemented them. We tested the algorithms on multiple samples and the experiment results are shown in the corresponding sections. For the DFA learning, we implemented Exbar and EDSM, two state of the art algorithms in their field. We benchmarked them with the same samples, and we compared and analysed the results. For the CFG learning, we implemented and benchmarked an algorithm for learning a grammar using a Minimally Adequate Teacher, introduced by Alexander Clark.

Based on our research and the experiment results, we may say that learning minimal DFAs from source code is possible and scalable to some extent for mainstream programming languages. Having a DFA, we can develop multiple testing tools that use state machines as software models, e.g. if we have some state-based model, we can make coverage related claims. Learning CFGs from source code alone (without having a partial grammar) is also possible, but does not scale to practical problems. There are techniques to overcome this limitation, e.g. Dubey et al. and Saha et al. [10, 14] propose algorithms that start with an incomplete grammar and add production rules to complete it.

The applications for grammatical inference are not within the scope of this thesis, and all presented scenarios are merely examples and sketches for future work. Grammatical inference is a mature topic and there are multiple published papers, that treat techniques and algorithms on how to induct grammars of different types of languages. For all that, there are many hidden assumptions in the published methods, which were made explicit in this thesis. Also, replication of these methods is not easy, because the authors do not provide access to their source code.

In the future, we would like to extend and optimize the implemented algorithms, and try to make them more practical for programming languages. One optimization could be to introduce tokens, to reduce the number of tree paths for DFAs, and to reduce the number of congruence classes for CFGs. We would also want to implement and analyse the algorithms that start with an incomplete grammar (e.g. the method presented by Dubey et al. [14]).

Bibliography

[1] Robert L. Glass. Frequently Forgotten Fundamental Facts About Software Engineering. *IEEE Softw.*, 18:112–111, 2001. URL http://dx.doi.org/10.1109/MS.2001.922739.

[2] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an Engineering Discipline for Grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14:331–380, 2005. URL http://doi.acm.org/10.1145/1072997.1073000.

[3] Andrew Stevenson and James R. Cordy. Grammatical Inference in Software Engineering: An Overview of the State of the Art. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, volume 7745, pages 204–223. Springer Berlin Heidelberg, 2013. URL http://dx.doi.org/10.1007/978-3-642-36089-3_12.

[4] E Mark Gold. Language identification in the Limit. *Information and Control*, 10(5): 447 – 474, 1967. URL http://www.sciencedirect.com/science/article/pii/S0019995867911655.

[5] Dana Angluin. Inductive Inference of Formal Languages from Positive Data. *Information and Control*, 45(2):117 – 135, 1980. URL http://www.sciencedirect.com/science/article/pii/S0019995880902855.

[6] Dana Angluin. Queries and Concept Learning. *Mach. Learn.*, 2(4):319–342, 1988. URL http://dx.doi.org/10.1023/A:1022821128753.

[7] L. G. Valiant. A Theory of the Learnable. *Commun. ACM*, 27(11):1134–1142, 1984. URL http://doi.acm.org/10.1145/1968.1972.

[8] Ming Li and Paul M. B. Vitányi. Learning Simple Concepts Under Simple Distributions. *SIAM JOURNAL OF COMPUTING*, 20:911–935, 1991. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.7234.

[9] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956. URL http://www.chomsky.info/articles/195609--.pdf.

[10] Diptikalyan Saha and Vishal Narula. Gramin: A System for Incremental Learning of Programming Language Grammars. In *Proceedings of the 4th India Software Engineering Conference*, pages 185–194. ACM, 2011. URL http://doi.acm.org/10.1145/1953355.1953380.

[11] Diptikalyan Saha. Gramin: A Programming Language Grammar Inference System, 2010. URL http://researcher.watson.ibm.com/researcher/files/in-diptsaha/gramin.pdf.

[12] Alpana Dubey, Pankaj Jalote, and SanjeevKumar Aggarwal. Inferring Grammar Rules of Programming Language Dialects. In Yasubumi Sakakibara, Satoshi Kobayashi, Kengo Sato, Tetsuro Nishino, and Etsuji Tomita, editors, *Grammatical Inference: Algorithms and Applications*, volume 4201, pages 201–213. Springer Berlin Heidelberg, 2006. URL http://dx.doi.org/10.1007/11872436_17.

[13] Alpana Dubey. Goodness Criteria for Programming Language Grammar Rules. *SIGPLAN Not.*, 41:44–53, 2006. URL http://doi.acm.org/10.1145/1229493.1229500.

[14] A. Dubey, P. Jalote, and S.K. Aggarwal. Learning context-free grammar rules from a set of program. *Software, IET*, 2, 2008. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4543987.

[15] Oscar Nierstrasz, Markus Kobel, Tudor Girba, Michele Lanza, and Horst Bunke. Example-Driven Reconstruction of Software Models. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 275–286. IEEE Computer Society, 2007. URL http://dx.doi.org/10.1109/CSMR.2007.23.

[16] Markus Kobel. Parsing by Example. Diploma thesis, University of Bern, 2005. URL http://scg.unibe.ch/archive/masters/Kobe05a.pdf.

[17] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75:87–106, 1987. URL http://dx.doi.org/10.1016/0890-5401(87)90052-6.

[18] Alexander Clark and Rémi Eyraud. Polynomial Identification in the Limit of Substitutable Context-free Languages. *J. Mach. Learn. Res.*, 8:1725–1745, 2007. URL http://dl.acm.org/citation.cfm?id=1314498.1314556.

[19] Alexander Clark. Distributional Learning of Some Context-Free Languages with a Minimally Adequate Teacher. In JoséM. Sempere and Pedro García, editors, *Grammatical Inference: Theoretical Results and Applications*, volume 6339, pages 24–37. Springer Berlin Heidelberg, 2010. URL http://dx.doi.org/10.1007/978-3-642-15488-1_4.

[20] L. Boasson and G. Senizergues. NTS languages are deterministic and congruential. *Journal of Computer and System Sciences*, 31, 1985. URL http://www.sciencedirect.com/science/article/pii/002200008590056X.

[21] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[22] Alexander Clark, Rémi Eyraud, and Amaury Habrard. A Polynomial Algorithm for the Inference of Context Free Languages. In Alexander Clark, François Coste, and Laurent Miclet, editors, *Grammatical Inference: Algorithms and Applications*, volume 5278, pages 29–42. Springer Berlin Heidelberg, 2008. URL http://dx.doi.org/10.1007/978-3-540-88009-7_3.

[23] Kevin J. Lang. Faster Algorithms for Finding Minimal Consistent DFAs. Technical report, 1999. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.7130.

[24] Arlindo L. Oliveira and Joao P. Marques-Silva. Efficient Search Techniques for the Inference of Minimum Size Finite Automata. In *In Proceedings of the 1998 South American Symposium on String Processing and Information Retrieval, Santa Cruz de La Sierra*, pages 81–89. IEEE Computer Society Press, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.4999.

[25] Kevin Lang, Barak Pearlmutter, and Rodney Price. Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.5084.

[26] S. Crespi-Reghizzi, M. A. Melkanoff, and L. Lichten. The Use of Grammatical Inference for Designing Programming Languages. *Commun. ACM*, 16, 1973. URL http://doi.acm.org/10.1145/361952.361958.

[27] Stefano Crespi-Reghizzi. Reduction of Enumeration in Grammar Acquisition. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, pages 546–552. Morgan Kaufmann Publishers Inc., 1971. URL http://dl.acm.org/citation.cfm?id=1622876.1622933.

[28] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[29] E Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302 – 320, 1978.

[30] Orlando Cicchello and Stefan C. Kremer. Inducing Grammars from Sparse Data Sets: A Survey of Algorithms and Results. *J. Mach. Learn. Res.*, 4:603–632, 2003. URL http://dx.doi.org/10.1162/153244304773936063.

[31] M. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., 1978.

[32] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley Longman Publishing Co., Inc., 1969.

[33] Miguel Bugalho and Arlindo L. Oliveira. Inference of Regular Languages Using State Merging Algorithms with Search. *Pattern Recogn.*, 38:1457–1467, 2005. URL http://dx.doi.org/10.1016/j.patcog.2004.03.027.

[34] François Coste and Jacques Nicolas. Regular Inference as a graph coloring problem. In *In Workshop on Grammar Inference, Automata Induction, and Language Acquisition*, pages 9–7, 1997. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.4048.

[35] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[36] Aaron Gorenstein. Great Algorithms: CYK, 2013. URL http://pages.cs.wisc.edu/~agorenst/cyk.pdf.