A Survey of Grammatical Inference in Software Engineering

Andrew Stevenson, James R. Cordy

School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6

Abstract

Grammatical inference – used successfully in a variety of fields such as pattern recognition, computational biology and natural language processing – is the process of automatically inferring a grammar by examining the sentences of an unknown language. Software engineering can also benefit from grammatical inference. Unlike these other fields, which use grammars as a convenient tool to model naturally occuring patterns, software engineering treats grammars as first-class objects typically created and maintained for a specific purpose by human designers. We introduce the theory of grammatical inference and review the state of the art as it relates to software engineering.

Keywords: grammatical inference, software engineering, grammar induction

---

1. Introduction

The human brain is extremely adept at seeing patterns by generalizing from specific examples, a process known as inductive reasoning. This is precisely the idea behind grammatical induction, also known as grammatical inference, where the specific examples are sentences and the patterns are grammars. Grammatical inference can be described concisely: "The learning task is to identify a "correct" grammar for the (unknown) target language, given a finite number of examples of the language" [74].

The main challenge of identifying a language of infinite cardinality from a finite set of examples is knowing when to generalize and when to specialize. Most inference techniques begin with the given sample strings and make a series of generalizations from them. These generalizations are typically accomplished by some form of state-merging (in finite automata), or non-terminal merging (in context-free grammars).

Email addresses: `andrews@cs.queensu.ca` (Andrew Stevenson), `cordy@cs.queensu.ca` (James R. Cordy)

Grammatical inference techniques are used to solve practical problems in a variety of different fields: pattern recognition, computational biology, natural language processing and acquisition, programming language design, data mining, and machine learning. Software engineering, in particular software language engineering, is uniquely qualified to benefit because it treats grammars as first-class objects with an intrinsic value rather than simply as a convenient mechanism to model patterns in some other subject of interest.

Historically there have been two main groups of contributors to the field of grammatical inference: theorists and empiricists. Theorists consider language classes and learning models of varying expressiveness and power, attempting to firm up the boundaries of what is learnable and how efficiently it can be learned, whereas empiricists start with a practical problem and, by solving it, find that they have made a contribution to grammatical inference research.

Grammatical inference is, intuitively as well as provably, a difficult problem to solve. The precise difficulty of a particular inference problem is dictated by two things: the complexity of the target language, and the information available to the inference algorithm about the target language. Naturally, simpler languages and more information both lead to easier inference problems. Most of the theoretical literature in this field investigates some specific combination of language class and learning model, and presents results for that combination.

In Section 2 we describe different learning models along with the type of information they make available to the inference algorithm. In Section 3 we explore the learnability, decidability, and computational complexity of different learning models applied to language classes of interest in software engineering: finite state machines and context-free grammars. Section 4 discusses the relationship between theoretical and empirical approaches, and gives several practical examples of grammatical inference in software engineering. In Section 5 we list the related surveys, bibliographies, and commentaries on the field of grammatical inference and briefly mention the emphasis of each. Finally, in Section 6 we discuss the main challenges currently facing software engineers trying to adopt grammatical inference techniques, and suggest future research directions to address these challenges.

This survey builds on our previous overview [75] by expanding Section 4, in particular the inference of grammars from execution traces. We also include a discussion on the benefits of GI to other grammar-based systems in Section 6.

2. Learning Models

The type of learning model used by an inference method is fundamental when investigating the theoretical limitations of an inference problem. This section covers the main learning models used in grammatical inference and discusses their strengths and weaknesses.

Section 2.1 describes *identification in the limit*, a learning model which allows the inference algorithm to converge on the target grammar given a sufficiently large quantity of sample strings. Section 2.2 introduces a teacher who knows the target language and can answer particular types of queries from the learner. This learning model is, in many cases, more powerful than learning from sample strings alone. Finally, Section 2.3 discusses the PAC learning model, an elegant method that attempts to find an optimal compromise between accuracy and certainty. Different aspects of these learning models can be combined and should not be thought of as mutually exclusive.

2.1. Identification in the limit

The field of grammatical inference began in earnest with E.M. Gold's 1967 paper, titled "Language Identification in the Limit" [34]. This learning model provides the inference algorithm with a sequence of strings one at a time, collectively known as a presentation. There are two types of presentation: positive presentation, where the strings in the sequence are in the target language; and complete presentation, where the sequence also contains strings that are not in the target language and are marked as such. After seeing each string the inference algorithm can hypothesize a new grammar that satisfies all of the strings seen so far, i.e. a grammar that generates all the positive examples and none of the negative examples. The term "information" is often used synonymously with "presentation" (e.g. positive information and positive presentation mean the same thing).

The more samples that are presented to the inference algorithm the better it can approximate the target language, until eventually it will converge on the target language exactly. Gold showed that an inference algorithm can identify an unknown language in the limit from complete information in a finite number of steps. However, the inference algorithm will not know when it has correctly identified the language because there is always the possibility the next sample it sees will invalidate its latest hypothesis.

Positive information alone is much less powerful, and Gold showed that any superfinite class of languages cannot be identified in the limit from positive presentation. A superfinite class of languages is a class that contains all finite languages and at least one infinite language. The regular languages are a superfinite class, indicating that even the simplest language class in Chomsky's hierarchy of languages is not learnable from positive information alone.

There has been much research devoted to learning from positive information because the availability of negative examples is rare in practice. However, the difficulty of learning from positive data is in the risk of overgeneralization, learning a language strictly larger than the target language. Angluin offers a means to avoid overgeneralization via "tell-tales", a unique set of strings that distinguish a language from other languages in its family [4]. She states conditions for the language family that, if true, guarantee that if the tell-tale strings are included in the positive presentation seen so far by the inference algorithm then it can be sure its current guess is not an overgeneralization.

2.2. Teacher and Queries

This learning model is similar in spirit to the game "twenty questions" and uses a teacher, also called an oracle, who knows the target language and answers queries from the inference algorithm. In practice, the teacher is often a human who knows the target language and aids the inference algorithm, but in theory can be any process hidden from the inference algorithm that can answer particular types of questions. Angluin describes six types of queries that can be asked of the teacher, two of which have a significant impact on language learning: membership and equivalence [8]. A teacher that answers both membership and equivalence queries is said to be a *minimally adequate teacher* because she is sufficient to help identify DFAs in polynomial time without requiring any examples from the target language [7].

For a membership query, the inference algorithm presents a string to the teacher who responds with "yes" if the string is in the language or "no" if it is not. Likewise for an equivalence query, the inference algorithm presents a grammar hypothesis to the teacher who answers "yes" or "no" if the guess is equivalent to the target grammar or not. In the case when the teacher answers "no" she also provides a counter-example, a string from the symmetric difference of the target language and the guessed language, allowing the inference algorithm to zero in on the target grammar. The symmetric difference of two sets A and B are the elements in either A or B but not both: $A \bigoplus B = (A \cup B) - (A \cap B)$.

Queries provide an alternate means to measure the learnability of a class of languages. They can be used on their own or in conjunction with a presentation of samples, either positive or complete, to augment the abilities of the learner. Section 3 discusses how learning with queries differs in difficulty from learning in the limit for various language classes.

2.3. PAC Learning

In 1984 Valiant proposed the Probably Approximately Correct (PAC) learning model [77]. This model has elements of both identification in the limit and learning from an oracle, but differs because it doesn't guarantee exact identification with certainty. As its name implies, PAC learning measures the correctness of its result by two user-defined parameters, ϵ and δ, representing accuracy and confidence respectively. This learning model is quite general and thus uses different terminology than typically found in formal languages, but of course applies just as well to grammatical inference. The goal is still to learn a "concept" (grammar) from a set of "examples of a concept" (strings).

Valiant assumes there exists a (possibly unknown) distribution D over the examples of a target concept that represent how likely they are to naturally occur, and makes available to the inference algorithm a procedure that returns these examples according to this distribution. As with Gold's identification in the limit, PAC learning incrementally approaches the target concept with more accurate guesses over time.

A metric is proposed to measure the distance between two concepts, defined as the sum of probabilities $D(w)$ for all w in the symmetric difference of $L(G)$ and $L(G')$. In Figure 1, the lightly shaded regions represent the symmetric difference between $L(G)$ and $L(G')$. The area of this region decreases as the distance between the two concepts decreases. In the case of grammatical inference,

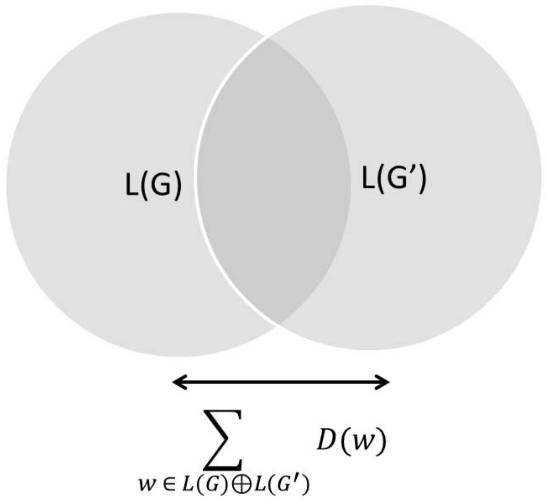

Figure 1: The PAC-learning measure of distance between two language concepts

these two concepts refer to the target grammar and the inference algorithm's current guess.

The PAC learning model's criteria for a successful inference algorithm is one that can confidently (i.e. with probability at least $1 - \delta$) guess a concept with high accuracy (i.e. distance to the target concept is less than ϵ). Valiant demonstrates the PAC learning model with a polynomial time algorithm that approximates bounded conjunctive normal form (k-CNF) and monotone disjunctive normal form (DNF) expressions using just the positive presentation from D and a membership oracle.

The novelty and uniqueness of Valiant's model intrigued the grammatical inference community, but negative and NP-hardness equivalence results (e.g. [46, 68]) dampened enthusiasm for PAC learning. Many feel Valiant's stipulation that the algorithm must learn polynomially under *all* distributions is too stringent to be practical since the learnability of many apparently simple concept classes are either known to be NP-hard, or at least not known to be polynomially learnable for all distributions.

Li and Vitanyi propose a modification to the PAC learning model that only considers simple distributions [57]. These distributions return simple examples with high probability and complex examples with low probability, where simplicity is measured by Kolmogorov complexity. Intuition is that simple examples speed learning. This is corroborated by instances of concepts given by the authors that are polynomially learnable under simple distributions but not known to be polynomially learnable under Valiant's more general distribution assumptions.

Despite the learnability improvements that simple PAC learning offers, the PAC learning model has attracted little interest from grammatical inference researchers in recent years. Identification in the limit and query-based learning models remain far more prevalent, with newer models such as neural networks and evolutionary algorithms also garnering interest.

3. Complexity

A significant portion of the grammatical inference literature is dedicated to an analysis of its complexity and difficulty, with results typically stated for a specific grammar class or type. The broadest form of result is simply whether a language class can be learned or not, while other results consider learning in polynomial time, learning the simplest grammar for the target language, or identifying the target language with a particular probability. Table 1 outlines the complexity results for different language classes and learning models.

Gold showed that a large class of languages can be identified in the limit from complete information including the regular, context-free, context-sensitive, and primitive recursive classes. This identification can be accomplished by a brute-force style of technique called *identification by enumeration* where, as each new example is presented, the possible grammars are enumerated until one is found that satisfies the presentation seen so far. By contrast, positive information alone cannot identify the aforementioned classes in the limit, nor any other superfinite class [34]. The subsequent sections describe the two easiest grammar classes to infer from the Chomsky hierarchy: regular grammars and context-free grammars. While small context-sensitive extensions to context-free grammars are common as features of source processing systems, little theoretical research has been attempted on the inference of context-sensitive and unrestricted grammars classes so they are omitted from this overview.

3.1. Deterministic Finite Automata

For any non-trivial language, multiple different grammars can be constructed to generate it. Likewise, there can exist DFAs that differ in their size but are equivalent in the sense that they accept the same language. When inferring a DFA from examples, it is naturally desirable to find the smallest DFA that accepts the target language. There exists only one such minimal DFA for a given language, known as the *canonical* DFA acceptor for that language. Despite the strong identification power of complete information, finding the minimal DFA that accepts an unknown regular language from a finite set of positive and negative samples is NP-complete [35].

Early claims of polynomial-time inference algorithms use the number of states in the target language's canonical acceptor as the input size. With this criteria, Angluin gives negative results for the polynomial identification of regular languages using membership queries only [5] or equivalence queries only [9]. However, if the membership oracle is augmented with a *representative sample* of positive data, a set of strings that exercise all the live transitions in the target language's canonical acceptor, then it is possible to identify a regular language in polynomial time [5]. By combining the power of both membership and equivalence queries, a regular language can be identified in polynomial time even

Language Class	**Presentation**		**Queries**		
	Complete	**Positive**	**Membership Only**	**Equivalence Only**	**Both**
Finite	Identifiable in the limit [34]	Identifiable in the limit [34]			
k-reversible automata		Polynomial [6]			
Strictly deterministic automata		Identifiable in the limit [83]			
Superfinite	Identifiable in the limit [34]	Not identifiable in the limit [34]			
Regular	Finding the minimum state DFA is NP-hard [35]		Polynomial for representative sample [5]	No polynomial algorithm [9]	Polynomial [7]
		Polynomial [18]			
Reversible context-free		Identifiable in the limit with structured strings [73]			
Noncounting context-free		Identifiable with structured strings [21]			
Very simple		Polynomial identifiable in the limit [82]		Polynomial [82]	
Structurally reversible context-free					Polynomial [14]
Simple deterministic					Polynomial [40]
Context-free			As hard as inverting RSA [10]		Polynomial with structured strings [72]

Table 1: Learnability and complexity results for various language classes using different learning models

without a single positive example in the unknown language [7]. Her proposed algorithm runs in time polynomial to the number of states in the minimum DFA and the longest counter-example provided by the equivalence oracle.

Several algorithms have been developed for the inference of DFAs from examples. These algorithms generally start by building an augmented prefix tree acceptor from positive and negative samples, then perform a series of state merges until all valid merges are exhausted. Each state merge has the effect of generalizing the language accepted by the DFA. The algorithms differ by how they select the next states to merge, constraints on the input samples, and whether or not they guarantee the inference of a minimal DFA.

An early state-merging algorithm is described by Trakhtenbrot and Barzdin that infers a minimal DFA in polynomial time, but requires that all strings up to a certain length are labeled as positive or negative [76]. The *regular positive and negative inference* (RPNI) algorithm also finds the canonical acceptor but allows an incomplete labeling of positive and negative examples which is more common in practice [67]. RPNI does, however, require the positive examples contain a characteristic set with respect to the target acceptor. A *characteristic set* is a finite set of positive examples $S \subset L(A)$ such that there is no other smaller automata A' where $S \subset L(A') \subset L(A)$. Lang provides convincing empirical evidence that his *exbar* algorithm out-performs comparable algorithms, and represents the state of the art in minimal DFA inference [51].

The algorithms discussed so far are guaranteed to infer a minimal DFA for the given examples, but *evidence driven state merging* (EDSM) algorithms relax this requirement for better scalability and performance. The order that states in a prefix tree are merged has a significant impact on an algorithm's performance because each merge restricts possible future merges. Bad merge decisions cause a lot of backtracking that can be avoided with smarter merge decisions. EDSM algorithms are so named because they use evidence from the merge candidates to determine a merge that is likely to be a good generalization, such as the heuristic proposed by Rodney Price in the first EDSM algorithm [52] and a winner of the Abbadingo Learning Competition. Differences in EDSM algorithms come down to the search heuristic used to select merges, and several have been tried such as beam search [51], stochastic search and the *self-adaptive greedy estimate* (SAGE) algorithm [45]. These search heuristics are comparable in performance and are the best known inference algorithms for large or complex DFAs.

3.2. Context-free grammars

Polynomial-time algorithms to learn higher grammar classes have also been investigated, in particular for context-free grammars. Unlike DFA inference, there is currently no known polynomial algorithm to identify a general context-free language from positive and negative samples, so most polynomial results in the literature either learn a strict subset of context-free grammars, use structured strings as input, or both.

Angluin and Kharitonov give a hardness result that applies to all context-free languages: constructing a polynomial-time learning algorithm for context-free grammars using membership queries only is computationally equivalent to cracking well-known cryptographic systems, such as RSA inversion [10].

Anecdotally, it appears a fruitful method to find polynomial-time learning algorithms for context-free languages from positive samples is to adapt corresponding algorithms from DFA inference, with the added stipulation that the sample strings be structured. A structured string is a string along with its unlabelled derivation tree, or equivalently a string with nested brackets to denote the shape of its derivation tree. Sakakibara has shown this method effective by adapting Angluin's results for learning DFAs by a minimally adequate teacher [7] and learning reversible automata [6] to context-free variants with structured strings [72, 73].

Clark et al. have devised a polynomial algorithm for the inference of languages that exhibit two special characteristics: the finite context property and the finite kernel property [18]. These properties are exhibited by all regular languages, many context-free languages, and some context-sensitive languages. The algorithm is based on positive data and a membership oracle. More recently, Clark has extended Angluin's result [7] of learning regular languages with membership and equivalence queries to a larger subclass of context-free languages [17].

Despite the absence of a general efficient context-free inference algorithm, many researchers have developed heuristics that provide relatively good performance and accuracy by sacrificing exact identification in all cases. We describe several such approaches related to software engineering in Section 4.

4. Applications in Software Engineering

Grammatical inference has its roots in a variety of separate fields, a testament to its wide applicability. Implementors of grammatical inference applications often have an unfair advantage over purely theoretical GI research because theorists must restrict themselves to inferring abstract machines (DFAs, context-free grammars, transducers, etc.) making no additional assumptions about the underlying structure of the data. Empiricists, on the other hand, can make many more assumptions about the structure of their data because their inference problem is limited to their particular domain.

Researchers attempting to solve a practical inference problem will usually develop their own custom solution, taking advantage of structural assumptions about their data. Often this additional domain knowledge is sufficient to overcome inference problems that theorists have proved impossible or infeasible with the same techniques in a general environment. The applications described in the following sections use grammatical inference techniques, but rarely result from applying a purely theoretical result to a practical problem.

4.1. Inference of General Purpose Programming Languages

Programming language design is an obvious area to benefit from grammatical inference because grammars themselves are first-class objects. Programming languages almost universally employ context-free, non-stochastic grammars to parse a program, which narrows the possible inference approaches considerably when looking for an inductive solution. When discussing the inference of programming language grammars here, the terms "sample" and "example" refer to instances of computer programs written in the target programming language.

Crespi-Reghizzi et al. suggest an interactive system to semi-automatically generate a programming language grammar from program samples [22]. This system relies heavily on the language designer to help the algorithm converge on the target language by asking for appropriate 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. 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. This latter task is significantly more difficult because the cognitive burden of creating correct missing sentences is much greater than identifying flaws in existing sentences. A modern technique to alleviate this burden is to mutate programs in the positive set to automatically produce examples for the negative set. The designer's corrections are finally fed back into the algorithm which corrects the grammar and outputs a new conjecture, and the process repeats until the target grammar is obtained.

Another system is proposed by Dubey et al. to infer a context-free grammar from positive samples for a programming language dialect when the standard language grammar is already known [29]. Their algorithm requires the non-terminals in the dialect grammar to be unchanged from the standard grammar, but allows for the terminals and production rules to be extended in the dialect grammar (i.e. new keywords can be added in the dialect along with their associated grammar rules). Their approach has the advantage of being fully automated so the designer simply needs to provide the dialect program samples and the standard language grammar. The authors have improved their original heuristic-based inference algorithm to a deterministic one that guarantees the output grammar parses all sample programs in the dialect language.

An alternate solution to the same dialect inference problem is described by Di Penta et al [28, 27]. Their approach is also automated and uses a genetic algorithm that appears to mimic the changes a human developer would reasonably make to the starting grammar. This has the advantage of keeping the starting grammar's non-terminal names and structure mostly intact in the resulting dialect grammar, allowing it to be understood and maintained by human developers with little disruption or rework.

4.2. Inference of Domain Specific Languages

Domain specific languages (DSLs) are languages whose syntax and notation are customized for a specific problem domain [64, 30, 33, 62], and are often more expressive and declarative compared to general purpose languages (GPLs). DSLs are intended to be used, and possibly designed, by domain experts who do not necessarily have a strong computer science background. Grammatical inference allows the creation of a grammar for a DSL by only requiring positive (and possibly negative) program samples by the designer.

The techniques for DSL inference are the same as those for GPLs, but far more success has been had inferring DSL grammars because of their smaller size and complexity. An empirical study corroborates the common intuition that DSL syntax is simpler than GPL syntax [20]. The authors of the study devised and applied a set of LR table metrics and generated language metrics applied to six DSLs (EXPR, FDL, EBNF, CFDG, GAL, ANTLR) and four GLPs (Ruby, C, Python, Java).

Črepinšek et al. have developed both a brute-force algorithm [79] and a genetic algorithm [78] to infer grammars for small DSLs using positive and negative samples. The brute-force approach exploits the fact that derivation trees of grammars in Chomsky Normal Form resemble full binary trees. These derivation trees are enumerated until one is found whose grammar parses all positive samples and no negative samples. The number of derivation trees explored can be drastically reduced by only searching for trees with a distinct nonterminal labelling, but even with this optimization the search space is still prohibitive for large programs.

For the evolutionary approach, the authors combine a set of grammar production rules into a *chromosome* representing a complete grammar, then apply crossover and mutation genetic operators that modify a population of chromosomes for the next generation. They use a fitness function that reflects the goal of having the target grammar accept all positive samples and reject all negative samples. Since a single random mutation is more likely to produce a grammar that rejects both positive and negative samples, the authors found that testing a chromosome on only positive samples converges more quickly to the target grammar than testing it on negative samples. Therefore, they chose a fitness value proportional to the total length (in tokens) of the positive samples that can be parsed by a chromosome. Negative samples, used to control overgeneralization, are only included in the fitness value if all positive samples are successfully parsed.

This genetic approach has been shown to accurately infer small DSLs [78], including one discussed by Javed et al. to validate UML class diagrams from use cases [41]. Javed et al. express UML class diagrams in a custom DSL and require a domain expert to provide positive and negative use cases written in that DSL. The system validates these use cases against the given UML diagrams and reports feedback to the user, who can use that feedback to change the UML diagrams to improve use case coverage. In this situation the computer is providing valuable context and information to the human user who is making the important generalization and specialization decisions for the grammar, but in theory UML diagrams can be synthesized entirely from the use case descriptions given a sufficiently powerful grammar inference engine.

Javed et al. extend their genetic algorithm by learning from positive samples only by using beam search and Minimum Description Length (MDL) heuristics [53] in place of negative examples to control overgeneralization of the conjectured grammar [43]. The idea here is to find the simplest grammar at each step and incrementally approach the target grammar. MDL is used as a measure of grammar simplicity, and beam search is used to more efficiently search the solution space of possible grammars. One disadvantage of this approach is it requires the positive samples to be presented in a particular order, from simplest to most complex, which allows the learning algorithm to encode the incremental differences from the samples into the target grammar. The authors' subsequent effort into a grammar inference tool for DSLs, called *MAGIc*, eliminates this need for an order-specific presentation of samples by updating the grammar based on the difference between successive (arbitrary) samples [61, 39]. This frees the designer from worrying about the particular order to present their DSL samples to the learning algorithm. Hrnčič et al. demonstrate how *MAGIc* can be adapted to infer DSLs embedded in a general purpose language (GPL) given the GPL's grammar [38]. The GPL's grammar rules are included in the chromosome, but frozen so they cannot mutate. Learning, therefore, occurs strictly on the DSL syntax and the locations in the GPL grammar where the embedded DSL is allowed.

The inference of DSLs can make it easier for non-programmer domain experts to write their own domain-specific languages by simply providing examples of their DSL programs. It can also be used in the migration or maintenance of legacy software whose grammar definitions are lost or unavailable.

4.3. Inference of Graph Grammars and Visual Languages

Unlike one-dimensional strings whose elements are connected linearly, visual languages and graphs are connected in two or more dimensions allowing for arbitrary proximity between elements. Graph grammars define a language of valid graphs by a set of production rules with subgraphs instead of strings on the right-hand side.

Fürst et al. propose a graph grammar inference algorithm based on positive and negative graph samples [32]. The algorithm starts with a grammar that produces exactly the set of positive samples then incrementally generalizes towards a smaller grammar representation, a strategy similar to typical DFA inference algorithms which build a prefix tree acceptor then generalize by merging states. The authors demonstrate their inference algorithm with a flowchart example and a hydrocarbon example, making a convincing case for its applicability to software engineering tasks such as metamodel inference and reverse engineering visual languages.

Another graph grammar inference algorithm is proposed by Ates et al. which repeatedly finds and compresses overlapping identical subgraphs to a single nonterminal node [11]. This system uses only positive samples during the inference process, but validates the resulting grammar by ensuring all the graphs in the training set are parsable and other graphs which are close to but distinct from the training graphs are not parsable. Ates et al. demonstrate their algorithm with two practical inferences: one for the structure of a programming language and one for the structure of an XML data source.

Kong et al. use graph grammar induction to automatically translate a webpage designed for desktop displays into a webpage designed for mobile displays [48]. The inference performed is similar to the aforementioned proposed by Ates et al. [11] because they both use the Spatial Graph Grammar (SGG) formalism and subgraph compression. The induction algorithm consumes webpages, or more accurately their DOM trees, to produce a graph grammar. After a human has verfied this grammar it is used to parse a webpage, and the resulting parse is used to segment the webpage into semantically related subpages suitable for display on mobile devices.

Graph grammar inference algorithms are less common than their text-based counterparts, but provide a powerful mechanism to infer patterns in complex structures. Parsing graphs is NP-hard in general, causing these algorithms to be more computationally expensive than inference from text. Most graph grammar learners overcome this complexity by restricting their graph expressiveness or employing search and parse heuristics to achieve a polynomial runtime.

4.4. Inference from Execution Traces

Grammatical inference is well suited to analyses involving program execution because the events in execution traces are temporally ordered in the same way that characters in strings are spatially ordered. Most of the applications described below use some form of encoding scheme to represent execution traces as strings before applying grammatical inference techniques to reverse engineer a model representing the behaviour of the running program.

Section 3 describes the difficulty inferring various language classes from positive samples alone, and in particular that only finite languages can be identified in the limit from positive samples [34]. The SEQUITUR algorithm, developed by Nevill-Manning and Witten, is designed to take a single string (long but finite) and produce a context-free grammar that reflects repetitions and hierarchical structure contained in that string [65]. This differs from typical grammar inference algorithms because it does not generalize. Data compression is an obvious use of this algorithm, but it has found other uses in software engineering. For example, Larus uses the SEQUITUR algorithm to concisely represent a program's entire runtime control flow and uses this dynamic control flow information to identify heavily executed paths in the program to focus performance and compiler optimization efforts [54].

Many of the inference algorithms discussed in this section use a variation of the k-tails NFA inference algorithm developed by Biermann and Feldman [12]. This algorithm takes an integer parameter $k = 1, 2, 3, \ldots$ and initially computes a set of strings of length k, called k-tails, for each state in the candidate NFA. The definition of a k-tail can be stated simply: "state a has a k-tail $t \in \Sigma^k$ iff starting from a it's possible to see the sequence t" [70]. Once k-tails are computed for each state, the finite state machine is generalized by merging states with identical k-tails until no more merges are possible.

Lo et al. compare algorithms, such as the k-tails algorithm, that infer simple finite state automata (FSA) representing a software system's behavioural model to algorithms that infer FSA annotated with data-flow information, known as "extended FSA" or EFSA [59]. Their comparison covers three general areas: 1) the quality of the inferred model, measured by how well it captures legal behaviour and rejects illegal behaviour, 2) the performance difference between FSA and EFSA during inference and analysis, and 3) how the effectiveness of these techniques differ under sparse or dense trace data.

Ammons et al. use runtime trace data to discover program specifications that can be subsequently refined and used for automated program verification [3]. The specifications in mind here are programming interfaces and abstract datatypes used by the application. For example, the inferred specification may capture the correct order of function calls in a socket connection lifecycle: create socket, bind to port, listen, accept connection, read/write bytes, close socket. Inference is done in two steps: (1) a probabilistic NFA learner [69] (based on the k-tail algorithm) produces an NFA with weighted transitions representing how often that transition was taken in the training set, then (2) a *corer* strips the low-frequency transitions and leaves the high-frequency "core", subject to some connectivity constraints. The result is an NFA that models common runtime behaviour, which the authors correlate with correct behaviour in a reasonably debugged program.

Reiss and Renieris propose a framework to capture trace data from a running program, then use several compaction schemes to represent the trace data as sequences [70]. In one such scheme function call traces are encoded by placing the callee to the right of its caller and using the "v" character as a placeholder for *return* (e.g. the function call trace "A calls B, A calls C" is encoded as "ABvCv" whereas the trace "A calls B, B calls C" is encoded as "ABCvv"). Some of these sequences are kept intact but compressed (e.g. with SEQUITUR) and others are generalized by recognizing repeating elements (e.g. the sequence AAAABABABC is represented as the regular expression $A^*(BA)^*BC$). The authors have developed a FSM construction algorithm designed to take advantage of the assumed structure in trace sequences such as frequent self-loops. The base algorithm merges FSM states that have at least one k-tail in common rather than requiring identical k-tails as in Biermann and Feldman's original formulation. This relaxed merging rule leads to a higher generalization of the final state machine.

A similar algorithm is described by Cook and Wolf to infer a finite state machine from a stream of execution events [19]. They modify the original k-tails algorithm to better handle the unrolling of loops and to ignore noise, defined as low-frequency event sequences. Viewing the trace data as an event stream allows for many different event sources to be incorporated into the inferred model, but also confounds the inference algorithm because unrelated software processes may be producing events concurrently which appear as interleaved events in the trace record. The authors acknowledge the problem of concurrency and suggest annotating each event with its source, or treating some event types as noise in the overall stream and correcting for it with their modified k-tail algorithm.

Jones and Oates directly address the problem of concurrent tasks producing an interleaved sequence of events and investigate how to disentangle them [44]. They restrict the problem to two instances of the same task (program) running concurrently, or in terms of automata theory: two executing instances of a single DFA whose transitions are traversed until both instances are in a final state. At each time step one of the two DFA instances is selected at random, and for that instance one of the valid transitions from the current state is selected at random. The symbol for the selected transition is appended to a common string shared by both instances and the next time step commences. The authors show a negative result for extracting a general DFA from its two-instance concurrent execution trace, but demonstrate a restricted language class called *terminated* languages where such an inference is possible. Terminated languages are distinguished by a special terminating character that appears at, and only at, the end of every valid string. The inference procedure involves building a super-DFA that accepts the interleaved language, then extracting the original DFA from it using a simple path-finding approach.

Walkinshaw et al. discuss how the most recent advances in grammatical inference theory can be used in the dynamic analysis of software [81]. Assuming the strings in the target language are once again program traces, they make the novel claim that dynamic analysis (runtime behaviour) is best suited to produce positive samples and static analysis is best suited to produce negative samples. Furthermore, learning with an oracle can be used by executing tests or performing static analysis to answer specific questions such as state reachability. In contrast to other domains these GI resources can be automatically generated and employed without a human's intervention or aid. In a series of experiments, the authors show the increasing effectiveness of adding negative information and an oracle to the inference process. They make a convincing case that combining dynamic analysis, static analysis, automatic test generation, and the latest in theoretical grammatical inference outperforms standard dynamic analysis techniques.

4.5. Other uses in Software Engineering

Cunha et al. attempt to automatically infer the functional dependencies between data cells in a spreadsheet to reverse engineer its underlying business model [23]. This model can be used to ensure the spreadsheet remains valid, consistent, and free from cell formula errors. The authors do not use inference techniques from the grammatical inference community, but rather use algorithms both of their own devising and related to reverse engineering database schema. They recognize that their inference heuristics are specific to their spreadsheet problem domain: "Note that several of these heuristics are possible only in the context of spreadsheets. . . In the spreadsheet domain rich context is provided, in particular, through the spatial arrangement of cells and through labels." [23]

Ahmed Memon proposes using grammatical inference in log files to identify anomalous activity [60]. He treats the contents of log files in a system running normally as a specific language, and any erroneous or anomalous activities reported in the log file are therefore not part of this language. Memon trains a grammar from positive log file samples of a system running normally, then parses subsequent log file entries using this grammar to identify anomalous activity. The inference procedure depends on knowledge of the domain, specifically sixteen text patterns that appear in typical log files: dates, times, IP addresses, session IDs, etc. Once these patterns are identified and normalized, a custom non-terminal merging algorithm is used to generalize the log file grammar.

Another recent use for grammatical inference is in the area of model-driven engineering. The relationship between a grammar and the strings it accepts is analogous to the relationship between a metamodel and the instance models it accepts. Javed et al. describe a method to use grammar inference techniques on a set of instance models to recover a missing or outdated metamodel [42]. The process involves converting the instance models in XML format to a domain-specific language, then performing existing grammar inference techniques on those DSL programs. The authors use their previously developed evolutionary approach [79] to do the actual inference, then recreate a metamodel in XML format from the result so the recovered metamodel can be loaded into a modeling tool. Liu et al. have recently extended this system to handle models with a more complex and segmented organizational structure [58]. The authors refer to these as multi-tiered domains because they support multiple viewpoints into the model.

Aartes et al. have proposed using grammatical inference to learn implementation details of a black-box software system by treating its inputs and associated outputs as a language to be learned [1]. Their algorithm applies a modified result from the theoretical literature, Angluin's L^* DFA learning algorithm with membership and equivalence queries. The black-box system to be learned acts as an oracle and can answer membership queries about its input-output pairs ($(input, output) \in L?$) by running the implementation on the input and seeing if the expected output is produced. Since this system cannot answer equivalence queries directly, the authors approximate this behaviour by first generating many random strings from the hypothesized grammar then ask a membership query for each generated string. If one of the strings is not in the target language it is returned as a counter-example. If all strings are in the language, the hypothesized grammar is assumed to be equivalent to the target grammar with some level of confidence. The more strings generated for an equivalence query, the higher the confidence.

Two similar problems are grammar convergence [50] and grammar recovery [49], both which involve finding grammars for a variety of software artifacts. The goal of grammar convergence is to establish and maintain a mapping between software artifact structures in different formats that embody the same domain knowledge. Grammatical inference can aid in the early steps of this process to produce a grammar for each knowledge representation by examining available concrete examples. Existing grammar transformation and convergence techniques can then be used on the resulting source grammars to establish a unified grammar.

Grammar recovery can be viewed as a more general version of the grammar inference problem because it seeks to recover a grammar from sources such as compilers, reference manuals and written specifications, in addition to concrete program examples. The effort by Lämmel and Verhoef to recover a VS COBOL II grammar includes leveraging visual syntax diagrams from the manual [49]. These diagrams give clues about the shape of the target grammar's derivation tree, knowledge that is known to greatly improve the accuracy of grammatical inference techniques. For example, reversible context-free and non-counting context-free languages are known to be identifiable from positive examples with these types of structured strings [73, 21]. Furthermore, structured strings can be used to identify any context-free language in polynomial time with a membership and equivalence oracle [72]. In the case of grammar recovery, an existing compiler for the language (even without the compiler source code) may be used as a membership oracle.

Nierstrasz et al. have more recently developed a tool to help recover a grammar, and ultimately a parser, from a collection of legacy source code [66]. Their technique requires the reverse engineer to identify text patterns corresponding to source elements (classes, methods, etc.), each of which will become productions in the resulting grammar. A parser is automatically generated from this grammar and validated against the legacy samples. Unparsable code is brought to the engineer's attention so he can provide further text patterns to cover those cases. The parser is regenerated and the process iterates until a sufficient amount of the legacy code can be parsed.

5. Related Surveys

Many surveys of grammatical inference have been written to introduce newcomers to the field and summarize the state of the art. Most give a thorough overview of grammatical inference in general, but each emphasise different aspects of the literature.

Fu and Booth (1986) give a detailed technical description of some early inference algorithms and heuristics with an emphasis on pattern recognition examples to demonstrate its relevance [31]. This survey is heavy on technical definitions and grammatical notation, suitable for someone with prior knowledge in formal languages who prefers to get right into the algorithms and techniques of grammatical inference.

Vidal (1994) provides a concise but thorough overview of the learning models and language classes in grammatical inference, with ample citations for follow-up investigation [80]. He presents each learning model in the context of fundamental learnability results in the field as well as their practical applications without getting too deeply into the details of each learning model.

Dana Ron's doctoral thesis (1995) on the learning of deterministic and probabilistic finite automata primarily investigates PAC learning as it relates to identifying DFAs, and describes practical applications of its use [71]. Although not exhaustive of grammatical inference in general, this thesis is a good reference for someone specifically interested in DFA inference.

Lee (1996) presents an extensive survey on the learning of context-free languages, including those that have non-grammar formalisms [55]. She discusses approaches that learn from both text and structured data, making it relevant to software engineering induction problems.

Sakakibara (1997) provides an excellent overview of the field with an emphasis on computational complexity, learnability, and decidability [74]. He covers a wide range of grammar classes including DFAs, context-free grammars and their probabilistic counterparts. This survey is roughly organized by the types of language classes being learned.

Colin de la Higuera (2000) gives a high-level and approachable commentary on grammatical inference including its historical progress and achievements [24]. He highlights key issues and unsolved problems, and describes some promising avenues of future research. This commentary is not meant as a technical introduction to inference techniques nor an exhaustive survey, and therefore contains no mathematical or formal notation. It rather serves as a quick and motivational read for anyone interested in learning about grammatical inference. A similar piece on grammatical inference is written by Honavar and de la Higuera (2001) for a special issue of the *Machine Learning* journal (volume 44), emphasizing the cross-disciplinary nature of the field [37].

Cicchello and Kremer (2003) and Bugalho and Oliveira (2005) survey DFA inference algorithms in depth, with excellent explanations about augmented prefix tree acceptors, state merging, the red-blue framework, search heuristics, and performance comparisons of state of the art DFA inference algorithms [16, 13].

de la Higuera (2005) provides an excellent guide to grammatical inference, geared toward people (not necessarily experts in formal languages or computational linguistics) who think grammatical inference may help them solve their particular problem [25]. He gives a general roadmap of the field, examples of how grammatical inference has been used in existing applications, and provides many useful references for further investigation by the reader.

Pieter Adriaans and Menno van Zaanen (2006) compare grammatical inference from three different perspectives: linguistic, empirical, and formal [2]. They introduce the common learning models and broad learnability results in the framework of each perspective, and comment on how these perspectives overlap. This survey is useful for someone who comes from a linguistic, empirical, or formal languages background and wishes to learn about grammatical inference.

6. Future Direction and Challenges

Although some software engineers are finding uses for grammatical inference it is still relatively rare in the field. The potential benefits of GI in software engineering extend beyond obvious applications, such as reverse engineering a parser, to other *grammar-based systems*. Mernik et al. catalogue grammar-based systems, defining them as "any system that uses a grammar and/or sentences produced by this grammar to solve various problems outside the domain of programming language definition and its implementation. The vital component of such a system is well structured and expressed with a grammar or with sentences produced by this grammar in an explicit or implicit manners" [63].

One such example of a grammar-based system is a grammatical approach to problem solving [36]. In this approach a problem description is coverted to a context-free attribute grammar where the grammar productions encode the problem domain concepts and the attributes capture the system behaviour. A sentence in the language of this grammar represents a use case of the system. Grammatical inference may be useful in this situation to work backward through the process: describe system use cases in some DSL, infer a grammar from the use case sentences, and finally generate a conceptual class diagram of the system from the inferred grammar.

Another grammar-based business application is discussed in [56], which uses a user-configurable context-free grammar to describe the workflow and interaction between system components. For example, two productions of the grammar presented in [56] are:

$$\begin{aligned} \mathit{OnlinePurchase} \rightarrow & [Identification], Presentation, Selection, \\ & Purchase, [Identification], Confirmation, \\ & OrderFulfillment \\ \mathit{ShoppingCartOperation} \rightarrow & \{\mathit{AddItem}|\mathit{DeleteItem}|\mathit{SaveCart}\}, \mathit{Checkout} \end{aligned}$$

The sentences of this language are implied by the execution sequence of methods in the various components of the running system. Grammatical inference techniques, especially those discussed in Section 4.4, could be used to recreate this grammar by analysing the execution of the system and reverse-engineer the system workflow at a relatively high level.

The theoretical work in grammatical inference is largely disconnected from these practical uses because implementers tend to use domain specific knowledge to craft custom solutions. Such solutions, while successful in some cases, usually ignore the powerful algorithms developed by the theoretical GI community. Domain knowledge should continue to be exploited – we are not advocating otherwise – but domain knowledge needs to be translated into a form general-purpose inference algorithms can use. We believe this is the biggest challenge currently facing software engineers wanting to use grammatical inference in their applications: how to map their domain knowledge to theoretical GI constraints.

Constraints can be imposed in several ways, such as simplifying the grammar class to learn, providing negative samples, adding a membership and/or equivalence oracle where none existed, or partially structuring the input data. Often these constraints are equivalent to some existing structural knowledge or implicit assumptions about the input data, but identifying these equivalences is nontrivial.

We motivate this approach with a concrete example, inspired by an example from Sakakibara [73]. Suppose you have a collection of computer programs written in an unknown language with a Pascal-like syntax and wish to infer a grammar from the collection. For clarity, Figure 2 shows a sample program in the collection and Figure 3 shows the target grammar to learn (a subset of the full Pascal grammar).

At first glance this inference problem seems too difficult to solve. It is a context-free grammar with positive samples only, and Gold proved learning a superfinite language in the limit from positive-only samples is impossible [34].

```
while limit > a do
    begin
        if a > max then max = a;
        a := a + 1
    end
```

Figure 2: A sample program in an unknown Pascal-like language

$$\begin{aligned}
Statement &\rightarrow Ident := Expression \\
Statement &\rightarrow \textbf{while}\ Condition\ \textbf{do}\ Statement \\
Statement &\rightarrow \textbf{if}\ Condition\ \textbf{then}\ Statement \\
Statement &\rightarrow \textbf{begin}\ Statementlist\ \textbf{end} \\
Statementlist &\rightarrow Statement; Statementlist \\
Statementlist &\rightarrow Statement \\
Condition &\rightarrow Expression > Expression \\
Expression &\rightarrow Term + Expression \\
Expression &\rightarrow Term \\
Term &\rightarrow Factor \\
Term &\rightarrow Factor \times Term \\
Factor &\rightarrow Ident \\
Factor &\rightarrow (Expression)
\end{aligned}$$

Figure 3: The target grammar for the Pascal-like language [73]

Even with the addition of negative samples there is no known algorithm to efficiently learn a context-free language.

On closer inspection, however, there is additional structural information in the input samples, hidden in a place grammarware authors and parsers are trained to ignore – the whitespace. By taking into account line breaks and indented sections of source code in the input samples, a structured string can be constructed for each program. If we further assume the target grammar is reversible then we can apply a result by Sakakibara, who showed reversible context-free grammars can be learned in polynomial time from positive structured strings [73].

This particular solution depends on two assumptions: (1) all the input samples have meaningful and consistent whitespace formatting, and (2) the target grammar is in the class of reversible context-free grammars. The assumption that the target grammar is reversible context-free is reasonable, as many DSLs would fit this criterion. The Pascal subset grammar in Figure 3 is in fact reversible context-free, but full Pascal is not because adding a production rule like $Factor \rightarrow Number$ to this grammar violates the criteria of reversibility [73].

Leveraging domain knowledge and structural assumptions is quite powerful when inferring grammars from examples and should be encouraged, but at present mapping this domain-specific knowledge to abstract constructs in grammatical inference research requires some creativity and awareness of theoretical results in the field. Walkinshaw et al.'s paper on dynamic analysis [81] discussed in Section 4.4 is an excellent example of this approach. Allowing the extensive work done in the theoretical grammatical inference community to bear on specific applications of GI would be a great boon to software engineering.

7. Conclusion

In grammatical inference, an inference algorithm must find and use common patterns in example sentences to concisely represent an unknown language in the form of a grammar. This process spans two axes of complexity: the language class to be learned and the learning model employed.

The theoretical foundations of grammatical inference are now well established thanks to contributions by Gold, Angluin and others. The state of the art, however, still has plenty of room to grow, and Colin de la Higuera identifies ten open problems on the theoretical side of grammatical inference that he believes are important to solve going forward [26].

In practice, assumptions can often be made which are not possible in a purely theoretical setting because a specific problem domain has limited scope, allowing for a better outcome than one would expect from simply applying the smallest enclosing theoretical result. Some work has already been done to investigate this relationship deeper, such as that by Kermorvant and de la Higuera [47] and Cano et al [15]. It would be valuable to find a widely applicable technique to equate domain assumptions with either a restriction in the class of language to learn, or an augmentation of the learning model.

Theoretical grammatical inference research continues to advance in many different directions: the language classes being learned, the learning models in use, the criteria for a successful inference, and the efficiency of the inference algorithms. Existing applications for grammatical inference are continually refined and new applications are found in a wide variety of disciplines.

References

[1] Aarts, F., Kuppens, H., Tretmans, G., Vaandrager, F., Verwer, S.: Learning and testing the bounded retransmission protocol. In Heinz, J., de la Higuera, C., Oates, T., eds.: JMLR Workshop and Conference Proceedings. Volume 21. (September 2012) 4–18

[2] Adriaans, P., van Zaanen, M.: Computational grammatical inference. Studies in Fuzziness and Soft Computing **194** (2006) 187–203

[3] Ammons, G., Bodk, R., Larus, J.R.: Mining specifications. In: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '02, New York, NY, USA, ACM (2002) 4–16

[4] Angluin, D.: Inductive inference of formal languages from positive data. Information and Control **45**(2) (1980) 117–135

[5] Angluin, D.: A note on the number of queries needed to identify regular languages. Information and Control **51**(1) (1981) 76–87

[6] Angluin, D.: Inference of reversible languages. Journal of the ACM (JACM) **29** (1982) 741–765

[7] Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75** (November 1987) 87–106

[8] Angluin, D.: Queries and concept learning. Machine Learning **2**(4) (1988) 319–342

[9] Angluin, D.: Negative results for equivalence queries. Machine Learning **5**(2) (July 1990) 121–150

[10] Angluin, D., Kharitonov, M.: When won't membership queries help? In: Proceedings of the twenty-third annual ACM symposium on Theory of computing. STOC '91, New York, NY, USA, ACM (1991) 444–454

[11] Ates, K., Kukluk, J., Holder, L., Cook, D., Zhang, K.: Graph grammar induction on structural data for visual programming. In: 18th IEEE International Conference on Tools with Artificial Intelligence, 2006. ICTAI '06. (November 2006) 232 –242

[12] Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE Transactions on Computers **21** (1972) 592–597 ACM ID: 1638997.

[13] Bugalho, M., Oliveira, A.L.: Inference of regular languages using state merging algorithms with search. Pattern Recogn. **38**(9) (September 2005) 1457–1467

[14] Burago, A.: Learning structurally reversible context-free grammars from queries and counterexamples in polynomial time. In: Proceedings of the seventh annual conference on Computational learning theory. COLT '94, New York, NY, USA, ACM (1994) 140–146

[15] Cano, A., Ruiz, J., García, P.: Inferring subclasses of regular languages faster using RPNI and forbidden configurations. In: Proceedings of the 6th International Colloquium on Grammatical Inference: Algorithms and Applications. ICGI '02, London, UK, Springer-Verlag (2002) 28–36

[16] Cicchello, O., Kremer, S.C.: Inducing grammars from sparse data sets: a survey of algorithms and results. J. Mach. Learn. Res. **4** (December 2003) 603–632

[17] Clark, A.: Distributional learning of some context-free languages with a minimally adequate teacher. In: Proceedings of the 10th international colloquium conference on Grammatical inference: theoretical results and applications. ICGI'10, Berlin, Heidelberg, Springer-Verlag (2010) 24–37

[18] Clark, A., Eyraud, R., Habrard, A.: A polynomial algorithm for the inference of context free languages. In: Proceedings of the 9th international colloquium on Grammatical Inference: Algorithms and Applications. ICGI '08, Berlin, Heidelberg, Springer-Verlag (2008) 29–42

[19] Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. **7**(3) (July 1998) 215–249

[20] Črepinšek, M., Kosar, T., Mernik, M., Cervelle, J., Forax, R., Roussel, G.: On automata and language based grammar metrics. Computer Science and Information Systems/ComSIS **7**(2) (2010) 309–329

[21] Crespi-Reghizzi, S., Guida, G., Mandrioli, D.: Noncounting context-free languages. Journal of the ACM (JACM) **25**(4) (October 1978) 571–580

[22] Crespi-Reghizzi, S., Melkanoff, M.A., Lichten, L.: The use of grammatical inference for designing programming languages. Communications of the ACM **16** (1973) 83–90

[23] Cunha, J., Erwig, M., Saraiva, J.: Automatically inferring ClassSheet models from spreadsheets. In: 2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). (September 2010) 93–100

[24] de la Higuera, C.: Current trends in grammatical inference. In: Proceedings of the Joint IAPR International Workshops on Advances in Pattern Recognition, London, UK, Springer-Verlag (2000) 28–31

[25] de la Higuera, C.: A bibliographical study of grammatical inference. Pattern Recognition **38** (September 2005) 1332–1348

[26] de la Higuera, C.: Ten open problems in grammatical inference. In Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E., eds.: Grammatical Inference: Algorithms and Applications. Volume 4201 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 32–44

[27] Di Penta, M., Lombardi, P., Taneja, K., Troiano, L.: Search-based inference of dialect grammars. Soft Computing **12**(1) (2008) 51–66

[28] Di Penta, M., Taneja, K.: Towards the automatic evolution of reengineering tools. In: Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on, IEEE (2005) 241–244

[29] Dubey, A., Jalote, P., Aggarwal, S.: Learning context-free grammar rules from a set of programs. IET Software **2**(3) (2008) 223–240

[30] Fowler, M.: Domain-specific languages. Pearson Education (2010)

[31] Fu, K.S., Booth, T.L.: Grammatical inference: introduction and survey\part i. IEEE Transactions on Pattern Analysis and Machine Intelligence **8** (May 1986) 343–359

[32] Fürst, L., Mernik, M., Mahnič, V.: Graph grammar induction as a parser-controlled heuristic search process, Budapest, Hungary (October 2011)

[33] Ghosh, D.: DSLs in action. Manning Publications Co. (2010)

[34] Gold, E.M.: Language identification in the limit. Information and Control **10**(5) (1967) 447–474

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

[36] Henriques, P.R., Kosar, T., Mernik, M., Pereira, M.J.V., Zumer, V.: Grammatical approach to problem solving. In: Information Technology Interfaces, 2003. ITI 2003. Proceedings of the 25th International Conference on, IEEE (2003) 645–650

[37] Honavar, V., de la Higuera, C.: Introduction. Machine Learning **44**(1–2) (2001) 5–7

[38] Hrnčič, D., Mernik, M., Bryant, B.R.: Embedding dsls into gpls: A grammatical inference approach. Information Technology and Control **40**(4) (December 2011)

[39] Hrnčič, D., Mernik, M., Bryant, B.R., Javed, F.: A memetic grammar inference algorithm for language learning. Applied Soft Computing **12**(3) (March 2012) 1006–1020

[40] Ishizaka, H.: Polynomial time learnability of simple deterministic languages. Machine Learning **5**(2) (July 1990) 151–164

[41] Javed, F., Mernik, M., Bryant, B.R., Gray, J.: A grammar-based approach to class diagram validation. (2005)

[42] Javed, F., Mernik, M., Gray, J., Bryant, B.R.: MARS: a metamodel recovery system using grammar inference. Inf. Softw. Technol. **50**(9-10) (August 2008) 948–968

[43] Javed, F., Mernik, M., Sprague, A., Bryant, B.: Incrementally inferring context-free grammars for domain-specific languages. Proceedings of the Eighteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'06) (2006) 363–368

[44] Jones, J., Oates, T.: Learning deterministic finite automata from interleaved strings. In: Proceedings of the 10th international colloquium conference on Grammatical inference: theoretical results and applications. ICGI'10, Berlin, Heidelberg, Springer-Verlag (2010) 80–93 ACM ID: 1886273.

[45] Juillé, H., Pollack, J.B.: A stochastic search approach to grammar induction. In: Proceedings of the 4th International Colloquium on Grammatical Inference. ICGI '98, London, UK, UK, Springer-Verlag (1998) 126–137

[46] Kearns, M., Li, M., Pitt, L., Valiant, L.: On the learnability of boolean formulae. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. STOC '87, New York, NY, USA, ACM (1987) 285–295

[47] Kermorvant, C., Higuera, C.D.L.: Learning languages with help. In: Proceedings of the 6th International Colloquium on Grammatical Inference: Algorithms and Applications. ICGI '02, London, UK, Springer-Verlag (2002) 161–173

[48] Kong, J., Ates, K., Zhang, K., Gu, Y.: Adaptive mobile interfaces through grammar induction. In: 20th IEEE International Conference on Tools with Artificial Intelligence, 2008. ICTAI '08. Volume 1. (November 2008) 133 –140

[49] Lämmel, R., Verhoef, C.: Semi-automatic grammar recovery. Softw. Pract. Exper. **31**(15) (December 2001) 1395–1448

[50] Lämmel, R., Zaytsev, V.: An introduction to grammar convergence. In: Proceedings of the 7th International Conference on Integrated Formal Methods. IFM '09, Berlin, Heidelberg, Springer-Verlag (2009) 246–260

[51] Lang, K.J.: Faster algorithms for finding minimal consistent DFAs. Technical report (1999)

[52] Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: Proceedings of the 4th International Colloquium on Grammatical Inference, London, UK, Springer-Verlag (1998) 1–12

[53] Langley, P., Stromsten, S.: Learning context-free grammars with a simplicity bias. Proceedings of the Eleventh European Conference on Machine Learning (2000) 220–228

[54] Larus, J.R.: Whole program paths. In: ACM SIGPLAN Notices. PLDI '99, New York, NY, USA, ACM (1999) 259–269

[55] Lee, L.: Learning of context-free languages: A survey of the literature. REP (1996) 12–96

[56] Levi, K., Arsanjani, A.: A goal-driven approach to enterprise component identification and specification. Commun. ACM **45**(10) (October 2002) 45–52

[57] Li, M., Vitányi, P.M.B.: Learning simple concepts under simple distributions. Siam Journal of Computing **20** (1991) 911–935

[58] Liu, Q., Bryant, B.R., Mernik, M.: Metamodel recovery from multi-tiered domains using extended MARS. In: Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference. COMPSAC '10, Washington, DC, USA, IEEE Computer Society (2010) 279–288

[59] Lo, D., Mariani, L., Santoro, M.: Learning extended FSA from software: An empirical assessment. J. Syst. Softw. **85**(9) (September 2012) 2063–2076

[60] Memon, A.U.: Log File Categorization and Anomaly Analysis Using Grammar Inference. Master of science, Queen's University (2008)

[61] Mernik, M., Hrnčič, D., Bryant, B., Sprague, A., Gray, J., Liu, Q., Javed, F.: Grammar inference algorithms and applications in software engineering. In: Information, Communication and Automation Technologies, 2009. ICAT 2009. XXII International Symposium on. (October 2009) 1–7

[62] Mernik, M.: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. Information Science Reference (2013)

[63] Mernik, M., Črepinšek, M., Kosar, T., Rebernak, D., Žumer, V.: Grammar-based systems: Definition and examples. Informatica **28**(3) (2004) 245–255

[64] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys (CSUR) **37** (2005) 316–344

[65] Nevill-Manning, C.G., Witten, I.H.: Identifying hierarchical structure in sequences: a linear-time algorithm. Journal of Artificial Intelligence Research **7**(1) (September 1997) 67–82

[66] Nierstrasz, O., Kobel, M., Girba, T., Lanza, M.: Example-driven reconstruction of software models. In: Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on, IEEE (2007) 275–286

[67] Oncina, J., García, P.: Identifying regular languages in polynomial time. In: Advances in Structural and Syntactic Pattern Recognition - Proceedings of the International Workshop on Structural and Syntactic Pattern Recognition, Bern, Switzerland (1992) 99–108

[68] Pitt, L., Valiant, L.G.: Computational limitations on learning from examples. Journal of the ACM (JACM) **35**(4) (October 1988) 965–984

[69] Raman, A., Patrick, J., North, P.: The sk-strings method for inferring PFSA. In: Proceedings of the workshop on automata induction, grammatical inference and language acquisition at the 14th international conference on machine learning (ICML97). (1997)

[70] Reiss, S.P., Renieris, M.: Encoding program executions. In: Proceedings of the 23rd International Conference on Software Engineering. ICSE '01, Washington, DC, USA, IEEE Computer Society (2001) 221–230

[71] Ron, D.: Automata Learning and its Applications. PhD thesis, Hebrew University (1995)

[72] Sakakibara, Y.: Learning context-free grammars from structural data in polynomial time. Theoretical Computer Science **76**(2-3) (November 1990) 223–242

[73] Sakakibara, Y.: Efficient learning of context-free grammars from positive structural examples. Information and Computation **97**(1) (1992) 23–60

[74] Sakakibara, Y.: Recent advances of grammatical inference. Theoretical Computer Science **185** (October 1997) 15–45

[75] Stevenson, A., Cordy, J.R.: Grammatical inference in software engineering: An overview of the state of the art. In Czarnecki, K., Hedin, G., eds.: SLE. Volume 7745 of Lecture Notes in Computer Science., Springer (2012) 204–223

[76] Trakhtenbrot, B.A., Barzdin, Y.M.: Finite Automata: Behaviour and Synthesis. North-Holland Publishing Company, Amsterdam (June 1973)

[77] Valiant, L.G.: A theory of the learnable. Communications of the ACM **27** (1984) 1134–1142

[78] Črepinšek, M., Mernik, M., Bryant, B.R., Javed, F., Sprague, A.: Inferring context-free grammars for domain-specific languages. Electronic Notes in Theoretical Computer Science **141**(4) (December 2005) 99–116

[79] Črepinšek, M., Mernik, M., Javed, F., Bryant, B.R., Sprague, A.: Extracting grammar from programs: evolutionary approach. ACM SIGPLAN Notices **40** (2005) 39–46

[80] Vidal, E.: Grammatical inference: An introductory survey. In Carrasco, R., Oncina, J., eds.: Grammatical Inference and Applications. Volume 862 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1994) 1–4

[81] Walkinshaw, N., Bogdanov, K., Holcombe, M., Salahuddin, S.: Improving dynamic software analysis by applying grammar inference principles. Journal of Software Maintenance and Evolution: Research and Practice **20**(4) (2008) 269–290

[82] Yokomori, T.: Polynomial-time learning of very simple grammars from positive data. In: Proceedings of the fourth annual workshop on Computational learning theory, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1991) 213–227

[83] Yokomori, T.: On polynomial-time learnability in the limit of strictly deterministic automata. Machine Learning **19**(2) (1995) 153–179