

# Parikh’s Theorem Made Symbolic

Matthew Hague<sup>1</sup>, Artur Jez<sup>2</sup>, and Anthony W. Lin<sup>3</sup>

<sup>1</sup> Royal Holloway University of London  
matthew.hague@rhul.ac.uk

<sup>2</sup> University of Wrocław  
aje@cs.uni.wroc.pl

<sup>3</sup> University of Kaiserslautern and Max-Planck Institute  
awlin@mpi-sws.org

## Abstract

Parikh’s Theorem is a fundamental result in automata theory with numerous applications in computer science. These include software verification (e.g. infinite-state verification, string constraints, and theory of arrays), verification of cryptographic protocols (e.g. using Horn clauses modulo equational theories) and database querying (e.g. evaluating path-queries in graph databases), among others. Parikh’s Theorem states that the letter-counting abstraction of a language recognized by finite automata or context-free grammars is definable in Linear Integer Arithmetic (a.k.a. Presburger Arithmetic). In fact, there is a linear-time algorithm computing existential Presburger formulas capturing such abstractions, which enables an efficient analysis via SMT-solvers. Unfortunately, real-world applications typically require large alphabets (e.g. Unicode, containing a million of characters) — which are well-known to be not amenable to explicit treatment of the alphabets — or even worse infinite alphabets.

Symbolic automata have proven in the last decade to be an effective algorithmic framework for handling large finite or even infinite alphabets. A symbolic automaton employs an effective boolean algebra, which offers a symbolic representation of character sets (i.e. in terms of predicates) and often lends itself to an exponentially more succinct representation of a language. Instead of letter-counting, Parikh’s Theorem for symbolic automata amounts to counting the number of times different predicates are satisfied by an input sequence. Unfortunately, naively applying Parikh’s Theorem from classical automata theory to symbolic automata yields existential Presburger formulas of exponential size. In this paper, we provide a new construction for Parikh’s Theorem for symbolic automata and grammars, which avoids this exponential blowup: our algorithm computes an existential formula in polynomial-time over (quantifier-free) Presburger and the base theory. In fact, our algorithm extends to the model of parametric symbolic grammars, which are one of the most expressive models of languages over infinite alphabets. We have implemented our algorithm and show it can be used to solve string constraints that are difficult to solve by existing solvers.

## 1 Introduction

Parikh’s Theorem [Parikh(1966)] (see also [Kozen(1997), Chapter H]) is a celebrated result in automata theory with far-reaching applications in computer science. These include software verification [Esparza and Ganty(2011), Hague and Lin(2011), Hague and Lin(2012)], decision procedures for array and string theories [Daca et al.(2016), Lin and Barceló(2016), Chen et al.(2020), Janku and Turonová(2019), Abdulla et al.(2019)], and evaluation and optimization of database queries [Barceló et al.(2012), David et al.(2012)], among others. Parikh’s Theorem concerns the so-called *letter-counting abstractions* of strings and languages. For example, the Parikh image of the string *abaacb* is the mapping  $f : \{a, b, c\} \rightarrow \mathbb{N}$ , where  $f(a) = 3$ ,  $f(b) = 2$ , and  $f(c) = 1$ .

In other words, the Parikh mapping abstracts away the ordering from a string (resp. a set of strings), i.e. yielding a multiset (resp. a set of multisets). Parikh's Theorem states that the class of context-free languages and the class of regular languages coincide modulo a Parikh mapping, both of which are moreover expressible as a formula in Linear Integer Arithmetic (a.k.a. Presburger Arithmetic). This is illustrated in the following example.

**Example 1.1.** *The Parikh image of the regular language  $L := (ab)^*$  is the set  $S$  containing all mappings  $f : \{a, b\} \rightarrow \mathbb{N}$  with  $f(a) = f(b)$ . Observe that  $S$  is also the Parikh image of the context-free language  $\{a^n b^n : n \geq 0\}$ . The Parikh image of  $L$  can be expressed as  $x_a = x_b \wedge x_a \geq 0$ , where  $x_a$  (resp.  $x_b$ ) represents the count for the letter  $a$  (resp.  $b$ ).*

Although the classical formulation of Parikh's Theorem concerns mainly the expressiveness of language models modulo taking Parikh images, its usefulness in applications was enabled only decades later by the development of efficient algorithms that compute an existential LIA formula (i.e. of the form  $\exists \bar{x} \varphi$ , where  $\varphi$  is quantifier-free) from a given automaton/grammar, enabling the exploitation of highly optimized SMT-solvers. In fact, building on the result by Esparza [Esparza(1997)], Verma et al. [Verma et al.(2005)] develops a linear-time algorithm that computes an existential LIA formula capturing the Parikh image of a given grammar. These results enabled the exploitation of Parikh's Theorem in many applications. Among others, these include verification of multithreaded programs with counters and possibly with (recursive) function calls [Hague and Lin(2011), Hague and Lin(2012), To(2009)], verification of concurrent and multithreaded programs [Esparza and Ganty(2011), Hague and Lin(2012)], verification of cryptographic protocols [Verma et al.(2005)], decision procedures for array theories [Daca et al.(2016)], decision procedures for string constraints [Lin and Barceló(2016), Chen et al.(2020), Janku and Turonová(2019), Abdulla et al.(2019)], query evaluation over graph databases [Barceló et al.(2012)], and reasoning over XML documents [David et al.(2012)]. The following two examples illustrate two simple applications of Parikh's Theorem for difficult problems.

**Example 1.2.** *The problem of checking emptiness of the intersection of several context-free languages has immediate applications in static analysis of concurrent programs (e.g. see [Bouajjani et al.(2003)]). However, since the problem of checking emptiness of two context-free languages is well-known to be undecidable (e.g. see [Kozen(1997)]), multiple incomplete methods are proposed, which include Parikh abstractions [Bouajjani et al.(2003)] and synthesis of regular separators/overapproximations [Gange et al.(2015), Gange et al.(2016), Long et al.(2012)], among others. Take the two context-free languages used in the benchmark of [Gange et al.(2016)]:*

$$L_1 := \{a^n c a^n : n > 0\} \quad L_2 := \{a^n c b^n : n > 0\}$$

*That  $L_1 \cap L_2 = \emptyset$  can be shown rather easily by considering the Parikh images  $S_1$  and  $S_2$  of, respectively,  $L_1$  and  $L_2$ . In fact,  $S_1$  contains precisely all multisets  $f$  with  $c \mapsto 1$ ,  $b \mapsto 0$ , and  $a \mapsto i$ , where  $i$  is positive even number. On the other hand,  $S_2$  contains precisely all multisets  $g$  with  $c \mapsto 1$ ,  $a, b \mapsto i$ , where  $i$  is a positive number. Thus, it follows that  $S_1 \cap S_2 = \emptyset$ . By employing the linear-time algorithm [Verma et al.(2005)] for Parikh images of context-free grammars and SMT-solvers, that  $S_1 \cap S_2 = \emptyset$  can be easily verified.*

**Example 1.3.** *String constraint solving is an area that has received a lot of attention, owing to their applications in the symbolic execution programs, for example in JavaScript [Saxena et al.(2010), Loring et al.(2019), Chen et al.(2022), Amadini et al.(2019)]. In this example, we deal with the*

simple string constraint

$$\varphi ::= zyx = xxz \wedge x \in a^* \wedge y \in a^+b^+ \wedge z \in b^*,$$

which the authors of a recent paper [Blahoudek et al.(2023)] have found to lead to failure for all string solvers that they have tried. We want to find a solution (i.e. mapping from the string variables  $x, y, z$  to strings over the alphabet  $\Sigma = \{a, b\}$ ) satisfying all the restrictions. The first restriction is an equation  $zyx = xxz$ , which enforces the two different concatenations of the strings instantiating the variables to be equal. For example, the mapping  $\lambda$ , where  $\lambda : z \mapsto aa$  and  $\lambda : y, x \mapsto a$ , satisfies this restriction; whereas the mapping  $\lambda$ , where  $\lambda : z \mapsto a$  and  $\lambda : y, x \mapsto b$ , does not. Each of the other restrictions is a regular constraint, which enforces a solution of a variable to satisfy certain regular patterns. For example,  $x \in a^*$  enforces that the variable  $x$  should be instantiated to be a string consisting only the letter  $a$ .

By using letter-counting abstraction, we can easily show the above example to be unsatisfiable. For each  $l \in \Sigma$ , let  $|x|_l$  denote the number of times  $l$  appearing in  $x$ . The letter-counting abstraction of the equation is the following quantifier-free LIA formula:  $\bigwedge_{l \in \Sigma} |x|_l + |y|_l + |z|_l = 2|x|_l + |z|_l$ , which can be simplified to  $\bigwedge_{l \in \Sigma} |y|_l = |x|_l$ . The letter-counting abstraction of the regular constraint  $x \in a^*$  (resp.  $z \in b^*$ ) is  $|x|_a \geq 0 \wedge |x|_b = 0$  (resp.  $|z|_a = 0 \wedge |z|_b \geq 0$ ). Finally, the letter-counting abstraction of the regular constraint  $y \in a^+b^+$  is  $|y|_a > 0 \wedge |y|_b > 0$ . Therefore, the letter-counting abstraction of  $\varphi$  is the quantifier-free LIA formula

$$\left( \bigwedge_{l \in \Sigma} |y|_l = |x|_l \right) \wedge |x|_a \geq 0 \wedge |x|_b = 0 \wedge |y|_a > 0 \wedge |y|_b > 0 \wedge |z|_a = 0 \wedge |z|_b \geq 0.$$

This formula is easily seen to be unsatisfiable since it asserts that  $0 = |x|_b = |y|_b > 0$ . Furthermore, this can be easily checked by virtually all existing SMT-solvers which support LIA (e.g. Z3 [de Moura and Björner(2008)]).

Despite the usefulness of Parikh's Theorem, most real-world applications require either large finite or even worse infinite alphabets, which renders the classical Parikh's Theorem impractical. For example, a regex  $r$  over UTF-16 — e.g.  $([\text{^}\text{x00}\text{--}\text{x7F}][\text{^}\text{x00}\text{--}\text{x7F}])^+$ , which accepts non-empty strings over non-ASCII characters of even length — has a total of  $2^{16}$  characters. A direct application of the linear algorithm from Verma et al. [Verma et al.(2005)] would yield a LIA formula with at least  $2^{16}$  variables, each keeping track of the count for each letter in UTF-16.

**Symbolic automata framework.** The framework of symbolic automata [D'Antoni and Veanes(2021), D'Antoni and Veanes(2017), Veanes et al.(2012)] (a.k.a. automata modulo theories) has proven in the last decade to be a fruitful approach for handling large finite or even infinite alphabets. The key to the framework is the symbolic representation of alphabets known as *effective boolean algebras*. Loosely speaking, an effective boolean algebra is a domain  $D$  with a class of monadic predicates (i.e. each has an interpretation as a subset of  $D$ ) that is closed under boolean operations (set-union, set-intersections, and set-complementation). The term “effective” refers to the fact that each monadic predicate  $P$  describes a *syntactic* property (e.g. a character class in Unicode, or a LIA formula with one free variable), and that checking whether the interpretation  $\llbracket P \rrbracket \subseteq D$  is empty is decidable. Many examples of effective boolean algebras are available including, notably, SMT algebras. For example, a LIA boolean algebra consists of domain  $D = \mathbb{Z}$  and monadic predicates of LIA (with existential quantifiers allowed), e.g.,  $P := x \equiv 0 \pmod{2}$ . The syntactic representation of predicates  $P$  and decidability of checking emptiness

can be taken advantage of by allowing an automaton transition of the form  $p \rightarrow_P q$ , where  $p, q$  are two automata states, representing all (potentially infinitely many) transitions of the form  $p \rightarrow_a q$ , where  $a$  satisfies  $P$ . Symbolic automata extend normal automata by allowing such transitions. An analogous representation of symbolic automata in terms of symbolic (regular) expressions [D'Antoni and Veanes(2021), Stanford et al.(2021)] is also possible, where predicates are allowed instead of concrete letters, e.g. the expression  $P^+$  represents the sequences of strings of odd numbers, whenever  $P := x \equiv 1 \pmod{2}$ .

Most analysis of symbolic automata is known to be reducible to the case of normal automata, but with an exponential blow-up in the alphabet size and the number of transitions [D'Antoni and Veanes(2017), Veanes et al.(2012)]. Although in most cases such an exponential blow-up is unavoidable in the worst case, clever algorithms that circumvent this exponential blow-up in practice have been devised on basic automata operations (e.g. boolean operations, transductions, learning, etc.). This takes us to the question of Parikh's Theorem in the setting of symbolic automata, which has so far not received much attention in the literature of symbolic automata.

A natural counterpart of letter-counting abstractions in the framework of symbolic automata is *predicate-counting abstractions*. Let us revisit Example 1.3 but with the letters  $a$  and  $b$  instantiated with different character classes. Let us start with  $a := \backslash d$  (meaning a digit) and  $b := \backslash D$  (meaning a non-digit). In this case, the predicate-counting abstraction with respect to  $a$  and  $b$  simply counts the numbers of occurrences of digits and non-digits in each string instantiations of  $x, y, z$ . The same reasoning used in Example 1.3 will allow us to prove unsatisfiability. On the other hand, consider  $a := \backslash s$  (including space symbols, tabs, and newlines) and  $b := .$  (meaning any character, except for a newline). Then,  $a$  (resp. the complement  $\bar{a}$ ) and  $b$  (resp.  $\bar{b}$ ) have non-empty intersections. (More precisely,  $a \cap b$  contains a space symbol,  $a \cap \bar{b}$  contains a newline character, and  $\bar{a} \cap b$  contains (say) a digit.) In general,  $n$  predicates in a symbolic automaton (equivalently, symbolic regular expression) can give rise to  $O(2^n)$  different “combinations” (a.k.a. *min-terms*). In other words, predicate-counting abstractions over symbolic automata can be reduced to letter-counting abstractions of normal automata, but *over an exponentially bigger alphabet*. Thus, the linear-time construction of Verma et al. for Parikh images of automata/grammars with (say) 14 predicates would yield already a large LIA formula with more than 15000 integer variables, which is very challenging to solve for existing SMT-solvers.

**Contributions.** Our main result is the first polynomial-time algorithm for computing an existential formula that captures the predicate-counting abstraction of a given symbolic automaton. In fact, the algorithm extends to more expressive formalisms, namely, symbolic context-free grammars even when they are extended with “read-only registers”; this is a model referred to as *parametric symbolic grammars*, which extend both symbolic automata [D'Antoni and Veanes(2021), D'Antoni and Veanes(2017)], symbolic visibly pushdown automata [D'Antoni and Alur(2014)], and symbolic variable/parametric automata [Grumberg et al.(2010), Faran and Kupferman(2020), Figueira et al.(2022)]. This new formalism has further applications including solving complex string constraints, e.g., with context-free constraints and, to some extent, the infamously difficult operator `to_re`, which converts strings to regular expressions. We have provided an implementation of our algorithm and demonstrated its efficacy in solving some difficult string constraints examples. We detail these contributions below.

As described above, the main technical difficulty of our problem is that the standard reduction from symbolic automata  $\mathcal{A}$  to normal automata  $\mathcal{A}'$  yields an exponential-sized alphabet, i.e.  $2^n$  when counting  $n$  predicates over theory  $T$ . This is in general not avoidable, e.g., symbolic

regular expressions of the form  $P_1 P_2 \cdots P_n$  over the LIA algebra, where  $P_i$  represents the set of all numbers that are congruent to 0 modulo the  $i$ th prime, have  $O(2^n)$  feasible min-terms. It turns out that, when considering predicate-counting abstractions, if  $w$  is accepted by  $\mathcal{A}$ , there is  $w'$  that is also accepted by  $\mathcal{A}$ , the predicate-counting abstractions of  $w$  and  $w'$  are the same and  $w'$  uses  $O((n + |\mathcal{N}|) \log(n + |\mathcal{N}|))$  different letters. Notice that this is an almost linear bound. In the case when  $\mathcal{A}$  is a parametric symbolic grammar, the size of the alphabet is  $O((n + |\mathcal{N}|) \log \ell(n + |\mathcal{N}|))$ , where  $\ell$  represents the maximum length of the right-hand side of a production in  $\mathcal{A}$ . Furthermore, we show that we can compute an existential  $T$ +LIA formula  $\varphi_{\mathcal{A}}$  that captures this predicate-counting abstractions. The formula  $\varphi_{\mathcal{A}}$  can be solved easily in the standard SMT framework of DPLL( $T$ , LIA) (e.g. [Kroening and Strichman(2008)]), which uses LIA and  $T$  solvers separately to add blocking lemmas. It follows immediately that we obtain decision procedures for analyzing satisfiability of predicate-counting abstractions (possibly restricted with additional LIA formulas) with a tight complexity upper bound: if  $T$  is NP-complete (resp. PSPACE-complete), then our problem is also NP-complete (resp. PSPACE-complete).

Since string constraints are defined over the Unicode alphabet<sup>1</sup>, one natural application of our result is in checking unsatisfiability of string constraints. By means of predicate-counting abstractions, we show how string constraints can be abstracted into the Parikh image of a symbolic grammar with an additional LIA restriction. Here, we allow an expressive and well-known subclass of string constraints (in particular, commonly used subclass of QF\_SLIA theory of SMT-LIB 2.6 [Barrett et al.(2017)]), which permits string concatenation, string equations, replace, regular constraints, contains, prefix-of, and suffix-of. Note that unsatisfiability of the latter implies unsatisfiability of the original string constraint, but not the converse; we are not aware of any non-trivial class for which the converse implication would hold, a trivial one is over a unary alphabet. At the same time our result admits an easy extension to sequence theories, which permit general effective boolean algebras (e.g. see [Jež et al.(2023)]).

We have implemented this translation, which takes an SMT-LIB file and produces a quantifier-free LIA formula, which can be easily checked using SMT-solvers. Our experimental results show that our procedure can substantially outperform existing string solvers for proving unsatisfiability (details are in Figure 1).

Finally, as mentioned above, our paper establishes Parikh's Theorem for generalizations of symbolic automata, i.e., parametric (symbolic) grammars and parametric (symbolic) push-down automata. Such formalisms are highly expressive, e.g., can express Dyck languages with *infinitely* many parenthesis symbols. This has many potential applications. The first application is the support of symbolic context-free constraints, i.e., an expression of the form  $x \in L$ , where  $L$  is given by a parametric grammar. In fact, classical results on string analysis (e.g. [Christensen et al.(2003), Minamide(2005)]) heavily use context-free constraints, which are not supported by SMT-LIB 2.6, but are supported by a handful of modern string solvers (e.g. TRAU [Abdulla et al.(2018)]). The second application is a partial support of a “future-looking” feature in SMT-LIB 2.6: `to_re`, which converts a string (possibly with variable names) to regular expressions. This is highly expressive, e.g., it allows encoding word equations with Kleene star like  $xy = z^*$ . Existing benchmarks allow only a very limited usage of `to_re`: strings with only constants (i.e. no variables) as input. Using parametric grammars, we can encode some interesting use cases of `to_re` beyond only string with constants, e.g., we can encode parametric regular constraints of the form  $x \in y^*$ , where both  $x$  and  $y$  are variables. Finally, parametric pushdown automata strictly extend the model of symbolic visibly pushdown automata [D'Antoni and Alur(2014)], which has applications to dynamic analysis of programs.

<sup>1</sup>See the SMT-LIB 2.6 specification <https://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml>

By allowing parameters and pushing values onto stack, our model allows some support of static analysis as well (of course with restrictions, for otherwise decidability would result).

**Organization.** We fix our notation and introduce our notion of parametric context-free grammar in Section 2. Also in this section, we show that these grammars admit a representation as parametric pushdown automata. In Section 3 we prove our new Parikh's Theorem for parametric symbolic grammars. In Section 4, we provide an abstraction of string constraint solving via predicate-counting constraints, and outline an extension to sequence theories. We describe our implementation and report our experimental results in Section 5. We conclude the paper in Section 6 with related work and future work.

## 2 Models

We start by introducing some basic notation for quantifier-free theories. Then we introduce parametric context-free grammars and finally an equivalent model of pushdown systems.

### 2.1 Preliminaries

For simplicity, we follow a model-theoretic approach to define our symbolic alphabets. This allows a more convenient treatment of *parameters* in our automata (e.g. see [Jež et al.(2023)]). Let  $\sigma$  be a set of vocabulary symbols. We fix a  $\sigma$ -structure  $\mathfrak{S} = (D; I)$ , where  $D$  can be a finite or an infinite set (i.e. the universe) and  $I$  maps each function/relation symbol in  $\sigma$  to a function/relation over  $D$ . The elements of our sequences will range over  $D$ . We assume that the quantifier-free theory  $T_{\mathfrak{S}}$  over  $\mathfrak{S}$  (including equality) is decidable. Examples of such  $T_{\mathfrak{S}}$  are abound from SMT, e.g., Linear Real Arithmetic and Linear Integer Arithmetic. We write  $T$  instead of  $T_{\mathfrak{S}}$ , when  $\mathfrak{S}$  is clear. Our quantifier-free formula will use *uninterpreted  $T$ -constants* (which will represent the parameters)  $a, b, c, \dots$ , and local variables  $x, y, z, \dots$ . We use  $\mathcal{C}$  to denote the set of all uninterpreted  $T$ -constants and  $\mathcal{V}$  to denote the set of all local variables.

We write  $\varphi(\mathcal{X}, \mathcal{Y})$  for a formula that is a Boolean combination of terms constructed from functions/relations in  $\sigma$ , uninterpreted constants  $\mathcal{X} \subseteq \mathcal{C}$ , and local variables  $\mathcal{Y} \subseteq \mathcal{V}$ . We say such a formula is a formula of  $T$ . An existential formula of  $T$  is of the form  $\exists x_1, \dots, x_n \cdot \varphi$ , where  $\varphi$  is a quantifier-free formula of  $T$ . An *interpretation* or *assignment* of the constants (resp. variables) is a map  $\mathcal{C} \rightarrow D$  (resp.  $\mathcal{V} \rightarrow D$ ). We write  $\varphi(X, Y)$  for the formula under these interpretations. We write  $T \models \varphi(X, Y)$  if  $\varphi$  is true in  $\mathfrak{S}$  under interpretations  $X, Y$ . A formula  $\varphi$  is satisfiable if there are interpretations  $X, Y$  such that the formula becomes true in  $\mathfrak{S}$ .

We write  $T_1 + T_2$  for the Boolean combination of theories  $T_1$  and  $T_2$ . That is, quantifier-free Boolean combinations of formulas  $\varphi$  where  $\varphi$  is a quantifier-free formula either of  $T_1$  or of  $T_2$ . When  $T_1$  and  $T_2$  satisfy certain conditions, decision procedures for  $T_1$  and  $T_2$  can be combined, e.g. with Nelson-Oppen [Nelson and Oppen(1979)].

### 2.2 Parametric Context-Free Grammars

We introduce the notion of *parametric context-free grammars*, which generalize context-free grammars to infinite alphabets in a similar way as parametric automata [Faran and Kupferman(2020), Figueira et al.(2022), Figueira and Lin(2022)] generalize finite automata; note that parametric automata in turn generalize both symbolic automata [D'Antoni and Veanes(2021), D'Antoni and Veanes(2017)] and variable automata [Grumberg et al.(2010)] and are expressible incomparable to symbolic register automata [D'Antoni et al.(2019)] (i.e. neither subsumes the other).



In brief, a production  $(A, \alpha, \varphi)$  of a parametric context-free grammar is an extension of a production  $(A, \alpha)$  of a classical context-free grammar. The production replaces a non-terminal  $A$ . However,  $\alpha$  is not a sequence of characters and non-terminals, but a sequence of local variables and non-terminals, e.g.  $yAy$ . The final component  $\varphi$  is a guard over both the parameters of the grammar and the local variables. E.g.  $\varphi$  may assert  $y = x$  for the parameter  $x$ . Hence, if  $x$  took the concrete value  $a$ , the production would replace  $A$  with  $aAa$ .

**Definition 2.1** (Parametric Grammar). *A parametric grammar over alphabet theory  $T$  is of the form  $\mathcal{G} = (\mathcal{X}, \mathcal{N}, D, P, S)$ , where  $\mathcal{X}$  is a finite set of parameters,  $\mathcal{N}$  are the nonterminal symbols,  $D$  is the (perhaps infinite) set of symbols from the domain,  $S$  is a starting symbol and  $P$  is a finite set of triples  $(A, \alpha, \varphi)$  from  $\mathcal{N} \times (\mathcal{Y} \cup \mathcal{N})^* \times T(\mathcal{X}, \mathcal{Y})$ , where  $\mathcal{Y}$  is the set of local variables.*

Here, *parameters* are uninterpreted  $T$ -constants, i.e.  $\mathcal{X} \subseteq \mathcal{C}$ ; the  $\mathcal{Y}$  variables are “local” in the sense that they are instantiated each time the production is used. Formulas that appear in the productions from  $P$  will be referred to as *guards*, since they restrict which symbols can be produced.

The semantics is defined by generalizing the rewriting-style definition of derivation for context-free grammars. A derivation begins with the singleton sequence  $\beta_0$  consisting only of the starting symbol  $S$ . From a sequence  $\beta A \beta'$  we can derive  $\beta \alpha' \beta'$ , when  $\alpha'$  can be derived from  $A$ , which is defined as follows: For interpretations  $X$  of  $\mathcal{X}$  and  $Y$  of  $\mathcal{Y}$  we write  $\alpha[\mathcal{X}/X, \mathcal{Y}/Y]$  for the substitution of the constants in  $\mathcal{X}$  by their interpretation under  $X$  and the variables  $\mathcal{Y}$  by their interpretation under  $Y$ . The nonterminal  $A$  can be rewritten by a rule  $(A, \alpha, \varphi(\mathcal{X}, \mathcal{Y}))$  with  $\alpha' = \alpha[\mathcal{X}/X, \mathcal{Y}/Y]$  when  $T \models \varphi(X, Y)$ . We call  $\alpha[\mathcal{X}/X, \mathcal{Y}/Y]$  an *instantiation* of  $\alpha$  and often denote it by  $\alpha'$ . We will often refer to a rule  $A \rightarrow \alpha$  in this case, suppressing the actual instantiation.

Then  $L_X(\mathcal{G})$  is the set of nonterminal-free sequences from  $D^*$  that can be derived by  $\mathcal{G}$  for an interpretation  $X$  of the parameters and  $L(\mathcal{G}) = \bigcup_X L_X(\mathcal{G})$ .

**Example 2.2.** *Consider a grammar  $\mathcal{G}$  with one nonterminal, no parameters and rules*

$$(S, ySy, \top), \quad (S, yy, \top)$$

where the formula  $\top$  is always true. Then the generated language is  $L(\mathcal{G}) = \{ww^R : w \in D^+\}$ . Let us add a single parameter  $x$  and change the rules to:

$$(S, yxSxy, \top), \quad (S, yxxy, \top).$$

Then the language is

$$L(\mathcal{G}) = \bigcup_{c \in D} \{d_1 c d_2 c \cdots d_k c c d_k c \cdots c d_1 : k \in \mathbb{N}_+, d_1, \dots, d_k \in D\}.$$

Notice that the parameter  $x$  takes the value  $c$  in all productions, but the local variable  $y$  can take a different value ( $d_i$ ) during each application of the productions.

Assuming  $D = \mathbb{N}$  we can add a condition on the production:

$$(S, yxSxy', "y > x \wedge y + y' = 0"), \quad (S, yxxy', "y > x \wedge y + y' = 0")$$

which, using  $(d_i)$  to denote the negative number  $-d_i$ , results in a language

$$\bigcup_{c \in D} \{d_1 c d_2 c \cdots d_k c c(d_k) c \cdots c(d_1) : k \in \mathbb{N}_+, d_1, \dots, d_k > c\}.$$

In proofs dealing the properties of the derivations (and not the derived sequences), we focus on rules  $A \rightarrow \alpha$  and their instantiations  $\alpha'$ , and not the guards, which are defined implicitly by the choice of rule.

Note that when all rules of the grammar are of the form  $A \rightarrow yB$  with  $|y| = 1$  and  $B \in \mathcal{N}$  or  $A \rightarrow \varepsilon$ , then we can think of  $\mathcal{G}$  as an automaton (with  $\mathcal{N}$  taking the role of states,  $S$  being the initial state and states such that  $B \rightarrow \varepsilon$  being the final states). Such an automaton is an extension of both symbolic automata [D'Antoni and Veanes(2021), D'Antoni and Veanes(2017), Veanes et al.(2012)] and variable automata [Grumberg et al.(2010), Faran and Kupferman(2020)], which is referred to as parametric (symbolic) automaton [Figueira and Lin(2022), Figueira et al.(2022), Jež et al.(2023)].

**Proposition 2.3.** *Assume that  $T$  is solvable in NP (resp. PSPACE). Then, deciding nonemptiness of a parametric grammar over  $T$  is in NP (resp. PSPACE).*

*Proof.* The proof is a generalization of a standard proof for context-free grammars and proofs for parametric automata, see [Faran and Kupferman(2020), Figueira et al.(2022), Figueira and Lin(2022)]. The language  $L(\mathcal{G})$  is nonempty if and only if it is nonempty for some parameters  $X$ , which we will existentially quantify over.

As in the case of standard context-free grammars, (for fixed parameters) the  $L_X(\mathcal{G})$  is nonempty if and only if we can order a subset of nonterminals  $A_0, A_1, \dots, A_k$ , where  $S = A_0$ , and for each find a rule  $(A_i, \alpha_i, \varphi_i)$  such that: if  $A_j$  is in  $\alpha_i$  then  $j > i$ ; and for each  $i$  the guard  $\varphi_i(X, Y)$  is satisfiable (for some  $Y$ ). Hence we guess the sequence  $A_0, A_1, \dots, A_k$ , verify the first condition and then verify the formula  $\exists X \cdot \bigwedge_{i=0}^k \exists Y \cdot \varphi_i(X, Y)$ . Clearly, if  $T$  is in NP (resp. PSPACE) then the above algorithm is in the same class.  $\square$

## 2.3 Parametric pushdown automata

As in the case of usual context-free grammars, there is a pushdown equivalent of our parametric grammar. Some care is needed, as some combinations of allowed transitions lead to a much more powerful model: for instance, it is not difficult to show that if we do not allow parameters then our grammar model cannot express the language  $\bigcup_{d \in D} d^*$ , yet it is easy to come up with a pushdown automaton which can recognize such languages (with no parameters): it is enough to store the first symbol on the stack and then compare each consecutive symbol with it; hence such a run should not be allowed. However, it seems natural that we should allow storing elements of  $D$  on the stack, so that, say, palindromes can be recognized. As a solution, if  $\mathcal{A}$  sees an element from  $D$  on the top of the stack then it must pop it from the stack. It cannot push and cannot make an  $\varepsilon$ -transition.

Nondeterministic parametric pushdown automata are formally defined as follows (cf. standard definition [Sipser(2013)]). We explain some restrictions on the definition below.

**Definition 2.4** (Nondeterministic Parametric Pushdown Automata). *A nondeterministic parametric pushdown automaton is a tuple  $(Q, D, \Gamma, \Delta, q_0, F)$ , where as usual  $Q$  is a finite set of states,  $\Gamma$  is finite stack alphabet, we require that  $\Gamma \cap D = \emptyset$ ,  $q_0 \in Q$  is the starting state,  $F \subseteq Q$*



is a set of accepting states and  $\Delta = \Delta_{\Gamma \cup \{\varepsilon\}} \cup \Delta_D \cup \Delta_{\varepsilon, \Gamma \cup \{\varepsilon\}} \cup \Delta_{\varepsilon, D}$  is a transition relation:

$$\begin{aligned}
\Delta_{\Gamma \cup \{\varepsilon\}} &\subseteq \underbrace{Q}_{\text{state}} \times \underbrace{(\Gamma \cup \varepsilon)}_{\text{stack top symbol}} \times \underbrace{T(\text{curr}, \mathcal{Y}, \mathcal{X})}_{\text{guard}} \times \underbrace{Q}_{\text{new state}} \times \underbrace{(\Gamma \cup \mathcal{X} \cup \mathcal{Y} \cup \text{curr})^*}_{\text{pushed to stack}} \\
\Delta_D &\subseteq \underbrace{Q}_{\text{state}} \times \underbrace{T(\text{curr}, \text{top}, \mathcal{Y}, \mathcal{X})}_{\text{guard}} \times \underbrace{Q}_{\text{new state}} \\
\Delta_{\varepsilon, \Gamma \cup \{\varepsilon\}} &\subseteq \underbrace{Q}_{\text{state}} \times \underbrace{(\Gamma \cup \varepsilon)}_{\text{stack top symbol}} \times \underbrace{T(\mathcal{Y}, \mathcal{X})}_{\text{guard}} \times \underbrace{Q}_{\text{new state}} \times \underbrace{(\Gamma \cup \mathcal{X} \cup \mathcal{Y})^*}_{\text{pushed to stack}} \\
\Delta_{\varepsilon, D} &\subseteq \underbrace{Q}_{\text{state}} \times \underbrace{T(\text{top}, \mathcal{Y}, \mathcal{X})}_{\text{guard}} \times \underbrace{Q}_{\text{new state}}.
\end{aligned}$$

Our definition is a strict extension of symbolic visibly pushdown automata [D'Antoni and Alur(2014)]. In the above definition *curr* is a variable bound to the symbol from  $D$  read by the automaton, *top* is a variable bound to stack top symbol, which is from  $D$ . The four cases of transition function are for ease of presentation, as in principle one could define  $\Delta$  as one set with some syntactic conditions. The case-distinction is as follows: In  $\Delta_{\Gamma \cup \{\varepsilon\}} \cup \Delta_D$  the automaton reads an input letter (bound to variable *curr*) and in  $\Delta_{\varepsilon, \Gamma \cup \{\varepsilon\}} \cup \Delta_{\varepsilon, D}$  it does not, i.e. those are  $\varepsilon$ -transitions and they do not refer to *curr* in the guards, nor in the word pushed to stack. In  $\Delta_{\Gamma \cup \{\varepsilon\}} \cup \Delta_{\varepsilon, \Gamma \cup \{\varepsilon\}}$  the stack topmost symbol is from  $\Gamma$  or we do not read the stack at all; in  $\Delta_D \cup \Delta_{\varepsilon, D}$  the stack top-most symbol is from  $D$ , in which case we are not allowed to push anything to the stack; on the other hand we can use it in the guard, say for comparison with *curr*. The reason for not allowing pushing to the stack in this case is that we do not want to copy the stack contents, which easily leads to recognition of language  $\cup_{d \in D} d^*$ , which should not be recognized without parameters.

Like the case of parametric grammars, the sequence pushed to the stack may in general depend on the read character *curr*, some local variables  $\mathcal{Y}$  and the parameter interpretation  $X$ . We require that when  $\alpha'$  is actually pushed to stack (say for a transition in  $\Delta_{\Gamma \cup \{\varepsilon\}}$ ) then  $\alpha' = \alpha[\text{curr}/d, \mathcal{X}/X, \mathcal{Y}/Y]$ , where  $d$  is the character read and  $Y$  is any assignment to  $\mathcal{Y}$  such that  $\varphi(d, a, X, Y)$  holds, where  $\varphi$  is the guard of the rule and  $a$  is the top of stack character;  $\alpha'$  for  $\varepsilon$ -transitions is defined similarly, i.e. as  $\alpha[\mathcal{X}/X, \mathcal{Y}/Y]$ .

Note, the guards can provide expressive power: E.g. for palindromes, we can store the first half of the read word on the stack and then, for the second half, check equality with the read symbol while popping character by character from the stack. That is, using the guard  $\text{top} = \text{curr}$ .

Let us describe the semantics, we will focus on  $\Delta_{\Gamma \cup \{\varepsilon\}}$ , the other cases are defined similarly. A configuration is a tuple  $(q, w)$  where  $q \in Q$  is a state and  $w \in (\Gamma \cup D)^*$  is the stack contents; as a convention, we assume that the stack top-most symbol is the first in  $w$ . Take a configuration  $(q, aw)$ , where  $a \in \Gamma \cup \varepsilon$ , and a transition  $(q, a, \varphi, q', \alpha)$ , let  $d$  be the symbol read by the automaton. If for some  $Y$  the  $\varphi(d, X, Y)$  holds then the automaton can change the configuration to  $(q', \alpha[\text{curr}/d, \mathcal{X}/X, \mathcal{Y}/Y]w)$ . Note that if  $a \in \Gamma$  then we need to pop it from the stack, and if  $a = \varepsilon$  then the transition does not depend on the stack contents. For  $\Delta_D$  the move is defined analogously, but when the stack contents is  $sw$  for  $s \in D$ , the guard is evaluated as  $\varphi(d, s, X, Y)$ . The semantics of  $\varepsilon$ -transitions is defined similarly.

A word  $w \in D^*$  (for parameter interpretation  $X$ ) is accepted if there is a run for  $w$  from  $(q_0, \varepsilon)$  to  $(q, w')$  for some  $w'$  and  $q \in F$ . By  $L_X(\mathcal{A})$  we denote the language recognized by a parametric NPDA  $\mathcal{A}$  for a given interpretation  $X$  of parameters, and define  $L(\mathcal{A}) = \bigcup_X L_X(\mathcal{A})$ .

**Theorem 2.5.** *The class of languages recognized by parametric context-free grammars and*

*parametric non-deterministic pushdown automata coincide.*

The equality is shown using two natural inclusions, proven in the Lemmata below.

**Lemma 2.6.** *Given a parametric grammar  $\mathcal{G}$  we can compute in polynomial-time a parametric NPDA  $\mathcal{A}$  of size linear in the size of  $\mathcal{G}$  such that for each parameter interpretation  $X$  we have  $L_X(\mathcal{G}) = L_X(\mathcal{A})$ .*

*Proof.* The proof is an adaptation of the classic proof, see [Sipser(2013), Lemma 2.21]; note that since  $\varepsilon$ -transitions are allowed, we do not need Greibach normal form, which is a little cumbersome in parametric setting.

Given a sequence  $w$  the  $\mathcal{A}$  will simulate the derivation of  $\mathcal{G}$  by always greedily expanding the left-most nonterminal and matching the left-most unmatched letter of the input sequence. We use the same parameter interpretation  $X$  as  $\mathcal{G}$  does. The automaton has three states  $q_0, q$ , and  $q_f$ . The starting state is  $q_0$  and  $q_f$  is the unique accepting state. The  $\Gamma$  is  $\mathcal{N} \cup \{\perp\}$ , where  $\perp$  represents the stack bottom. In  $q_0$ , the automaton  $\mathcal{A}$  pushes  $S\perp$  to the stack (so  $S$  is top-most) and moves to  $q$ , here  $S$  is the starting symbol of the grammar. In  $q$  if  $\mathcal{A}$  sees  $\perp$  on the stack then it moves to  $q_f$  and accepts (it cannot proceed). Otherwise, if the topmost symbol is  $A \in \mathcal{N}$  then  $\mathcal{A}$  chooses (nondeterministically) a rule  $A \rightarrow \alpha$  and its (valid) instantiation  $\alpha'$ , pops  $A$  and pushes  $\alpha'$  to the stack. If the topmost symbol is  $d \in D$  and the next symbol is  $d$  (that is  $top = curr$ ) then  $\mathcal{A}$  pops the letter and reads the next symbol from the input (and stays in  $q$ ). It is easy to see that the resulting automaton  $\mathcal{A}$  has size linear in  $\mathcal{G}$ , and can furthermore be computed in polynomial-time. A slight modification of a standard proof shows the for each instantions of parameters  $X$  we have  $L_X(\mathcal{G}) = L_X(\mathcal{A})$ , so also  $L(\mathcal{G}) = L(\mathcal{A})$ , as claimed.  $\square$

The other direction is slightly more involved. Our construction is exponential in the maximum length of  $\alpha$  appearing in a transition  $(q, c, \varphi, q', \alpha) \in \Delta_{\Gamma \cup \varepsilon}$  or in a transition  $(q, \varphi, q', \alpha) \in \Delta_{\varepsilon, \Gamma \cup \varepsilon}$ . Unlike the non-parametric case, we cannot split pushing transitions so that each transition pushes at most one symbol to the stack. This is because the values of  $\mathcal{Y}$  and  $curr$  cannot be transferred across separate transitions. Thus a rule pushing  $\alpha$  to the stack needs an exponential number of productions to handle all sequences of intermediate states that may occur while  $\alpha$  is later being popped. However, so long as we fix the maximum length of such  $\alpha$ , the resulting grammar is of polynomial size and can be computed in polynomial-time.

**Lemma 2.7.** *Given a parametric NPDA  $\mathcal{A}$  there is a parametric grammar  $\mathcal{G}$  such that for each parameter interpretation  $X$  we have  $L_X(\mathcal{G}) = L_X(\mathcal{A})$ . The size of  $\mathcal{G}$  is exponential in the maximum number  $M$  of push symbols appearing in any transition of  $\mathcal{A}$ . When  $M$  is fixed, the algorithm runs in polynomial-time.*

*Proof.* We modify a standard construction cf. [Sipser(2013), Lemma 2.27]. For a fixed parameter interpretation  $X$ ,  $L_X(\mathcal{A}_{q,q'})$  is the language recognized by  $\mathcal{A}$  with starting state  $q$ , final state  $q'$ . We modify  $\mathcal{A}$  so that:

- it has a single accepting state
- it empties the stack before accepting
- in each move it either pushes something (perhaps the empty sequence) to the stack or pops from the stack, but not both.

The first two conditions are easy to ensure, the last depends on the form of the transition:

- If the transition is from  $\Delta_D \cup \Delta_{\varepsilon, D}$  then it does not push, as required.

- If the transition is from  $\Delta_{\Gamma \cup \varepsilon} \cup \Delta_{\varepsilon, \Gamma \cup \varepsilon}$  and it reads  $\varepsilon$  from the stack, then it does not pop from the stack, as required.
- If the transition is from  $\Delta_{\Gamma \cup \varepsilon} \cup \Delta_{\varepsilon, \Gamma \cup \varepsilon}$  for a topmost symbol  $\gamma \in \Gamma$  then we create a new state  $q_{q, \gamma}$  and create an  $\varepsilon$ -transition from  $q$  that removes  $\gamma$  without reading a letter and goes to  $q_{q, \gamma}$ . Then from  $q_{q, \gamma}$  the automaton ignores the stack and acts as if it were in  $q$  with  $\gamma$  on top of the stack.

The defined automaton recognizes the same language (for parameter interpretation  $X$ .)

Note, that the conditions on the transition relation when the topmost symbol is from  $D$  are tailored so that the above separation of popping and pushing is possible.

In a standard proof of equivalence of NPDAs and CFGs, cf [Sipser(2013), Lemma 2.27], the computation of  $\mathcal{A}$  is split into parts in which it empties the stack (from the symbols it introduced). The assumption that  $\mathcal{A}$  can push at most one element to the stack makes the proof easier; however, we need to push more symbols. But this only means that when  $a_1 a_2 \cdots a_k$  is pushed to the stack, the computation is split into  $k$  subcomputations, in which it removes  $a_1, a_2, \dots, a_k$  from the stack.

To be more precise, let  $\Gamma = \mathcal{N} \times Q \times Q$  and denote its elements by  $A_{q, q'}$ , with the intention that  $L_X(A_{q, q'})$  is the language of words such that  $\mathcal{A}$  starting in  $q$  (and empty stack) will go on this word to the empty stack and state  $q'$ . In particular, we will set  $A_{q_0, q_f}$  as the starting symbol, where  $q_0$  is the starting state and  $q_f$  the unique final state, and then  $L(A_{q_0, q_f}) = L(\mathcal{A})$ .

Recall the classic construction, in which case each rule pushing an element to the stack pushes at most one symbol. When describing the computation taking the automaton from  $q$  to  $q'$  and from the empty stack to the empty stack, i.e. corresponding to the nonterminal  $A_{q, q'}$ , either the stack is emptied somewhere on the way, say at state  $q''$ , which means that we have a rule  $A_{q, q'} \rightarrow A_{q, q''} A_{q'', q'}$ , or it is emptied at the last step. Hence if the first transition pushes  $s$  to the stack, the last removes it from the stack and in the meantime the computation is as if it started and ended on an empty stack. Hence the rule is of the form  $A_{q, q'} \rightarrow a A_{p, p'} b$  such that there is a transition from  $q$  to  $p$ , reading  $a$ , pushing  $s$  and a transition from  $p'$  to  $q'$  popping  $s$ , reading  $b$  (both  $a, b$  can be a letter or  $\varepsilon$ ).

In our case we cannot assume that a transition pushes just a single symbol to the stack, as the sequence pushed may contain the same local variable or *curr* several times and splitting into many rules would lose this connection. Hence we need to consider rules pushing, say,  $s_1 \cdots s_k$  and the  $k$  rules that pop those letters (and in between  $\mathcal{A}$  acts as if it were starting and ending on the empty stack), at the same time and “compound” their computation.

We include the rules given below; The first bullet point states that a non-terminal  $A_{q, q}$  can be rewritten to  $\varepsilon$ . That simulates a move from  $q$  to  $q$  without firing any transitions. In the second bullet, the production  $A_{q, q''} \rightarrow A_{q, q'} A_{q', q''}$  handles the case where the stack is emptied at state  $q'$  on the run from  $q$  to  $q''$ .

- $A_{q, q} \rightarrow \varepsilon$
- for each  $q, q', q'' \in Q$  a rule

$$A_{q, q''} \rightarrow A_{q, q'} A_{q', q''}$$

- for each  $(q_0, \varepsilon, \varphi_0, q_1, s_1 \cdots s_k) \in \Delta_{\Gamma \cup \varepsilon} \cup \Delta_{\varepsilon, \Gamma \cup \varepsilon}$  (so the one pushing  $s_1 \cdots s_k$  to the stack and not looking at the stack) such that there are  $q_2, \dots, q_{k+1}$  (states in which  $s_1, \dots, s_k$  are popped) such that for all  $2 \leq i \leq k$  we have  $(q_i, \varphi_i, q_{i+1}) \in \Delta_D \cup \Delta_{\varepsilon, D}$  or  $(q_i, s'_i, \varphi_i, q_{i+1}, \varepsilon) \in \Delta_{\Gamma \cup \varepsilon} \cup \Delta_{\varepsilon, \Gamma \cup \varepsilon}$  (the transitions popping those letters) we add a production

$$A_{q_0, q_{k+1}} \rightarrow y_0 A_{q_1, q_2} y_1 A_{q_2, q_3} y_2 \cdots A_{q_k, q_{k+1}} y_k$$

and a guard

$$\varphi_0[curr/y_0, \mathcal{X}, \mathcal{Y}/\mathcal{Y}_0] \wedge \bigwedge_{i=1}^k \varphi_i[curr/y_i, top/s_i, \mathcal{X}, \mathcal{Y}/\mathcal{Y}_i]$$

the  $y_i$  is the symbol read by the  $i$ -th transition,  $\mathcal{Y}_i$  are the local variables of the  $i$ -th transition,  $s'_i$  is the symbol popped by  $i$ -th transition and  $\varphi_i$  is the guard of the  $i$ -th transition; those are described in detail below. Note that if the  $i$ -th transition for  $0 \leq i \leq k$  is an  $\varepsilon$ -transition then  $\varphi_i$  does not depend on the  $curr$ , and if it does not read the stack top element, then  $\varphi_i$  does not use  $top$ , but we write like this to streamline the argument.

Concerning  $y_i$ , corresponding to letters read by popping transitions, if the  $i$ -th transition is an  $\varepsilon$  transition then  $y_i = \varepsilon$  and otherwise it is a fresh local variable (which appears in the guard). Concerning  $s'_i$ , they “should be”  $s_i$ , but the problem arises when  $s_i$  is equal to  $curr$ , i.e.  $s_i$  pushed to the stack is the read letter; then  $curr$  “should be”  $y_0$ . Hence, if  $s_i = curr$  we define  $s'_i = y_0$ . That is, at  $q_0$  the read symbol  $y_0$  was pushed to the stack. Otherwise — if  $s_i \neq curr$  — we set  $s'_i = s_i$ , which could be both an element from  $\Gamma$  or  $D$ .

We have different fresh copies  $\mathcal{Y}_i$  of the local variables. This is because multiple pushdown transitions are encoded in a single rule. The value of the local variables may differ for each transition fired. Hence, we need separate copies for the different guards  $\varphi_i$  combined in the grammar production. Notice that these separate copies are never pushed onto the stack as there is only one pushing transition represented by the production.

The proof is now a generalization of the standard one [Sipser(2013), Lemma 2.27]. When starting from  $q$  with an empty stack and ending in  $q'$ , then if we reach the empty stack somewhere on the way we use  $A_{q,q''} \rightarrow A_{q,q'} A_{q',q''}$ . Otherwise, we push some  $s_1 \cdots s_k$  to the stack and take them from the stack one by one: i.e. for each  $i$  there is the first moment when  $s_i$  is taken from the stack and from the moment we took  $s_{i-1}$  right before we take  $s_i$  the  $\mathcal{A}$  acts as if on empty stack.  $\square$

### 3 Computing Parikh images

We first generalise the notion of a Parikh image to the parametric setting. Then we discuss the construction of Parikh images, and the complexity of a related decision problem.

#### 3.1 Definition

We first recall the classical definition of the Parikh image of a language. Take a finite alphabet  $\Sigma = \{a_1, \dots, a_n\}$ . For a word  $w \in \Sigma^*$ , let  $|w|_a$  be the number of occurrences of the character  $a$  in the word  $w$ . For a linearisation  $a_1, \dots, a_n$  of the characters of  $\Sigma$ , the Parikh image  $P(w)$  of  $w$  is a mapping  $f : \{1, \dots, n\} \rightarrow \mathbb{N}$  such that  $f(i) = |w|_{a_i}$  for all  $1 \leq i \leq n$ . For a language  $L \subseteq \Sigma^*$  the Parikh image is the set of mappings  $\{P(w) : w \in L\}$ . That is, the Parikh image counts the number of occurrences of each character in each word of  $L$ .

Parametric context-free grammars may have large or even infinite alphabets. Counting each of the characters is either impractical or impossible. Hence, we define a version of the Parikh image that is relative to a sequence of predicates. We then count the number of characters satisfying each predicate, rather than the number of each individual character.

**Definition 3.1** (Parametric Parikh Image). *For a word  $w = a_1 \dots a_k \in D^*$  and  $T$ -formula  $\psi(curr)$  over one local variable  $curr$ , the count  $|w|_\psi$  is the number of positions  $i$  of  $w$  such that*

$T \models \psi(a_i)$ . For a sequence  $\Psi := \psi_1, \dots, \psi_n$  of  $T$ -formulas, the Parikh image  $P_\Psi(w)$  of  $w$  over  $\Psi$  is a mapping

$$f : \{1, \dots, n\} \rightarrow \mathbb{N}$$

with  $f(i) = |w|_{\psi_i}$  for all  $i$ . For a parametric grammar  $\mathcal{G}$  over  $T$ , the Parikh image of  $\mathcal{G}$  over  $\Psi$  is

$$P_\Psi(L(\mathcal{G})) := \{P_\Psi(w) : w \in L\}.$$

Let us fix a parametric grammar  $\mathcal{G}$  over  $T$  and a sequence  $\Psi := \psi_1, \dots, \psi_n$  of  $T$ -formulas.

### 3.2 Representing the Parikh Image

We can construct a formula in the combined theory of  $T$  and quantifier-free Presburger arithmetic that represents the Parikh image of a given parametric grammar  $\mathcal{G}$ . Such a formula is polynomial in size and can be used as part of a query to an SMT solver to solve decision problems over the grammar. Below, let QFPresburger be the theory of quantifier-free fragment of Presburger arithmetic.

**Theorem 3.2.** *Let  $\mathcal{G}$  be a parametric grammar over the theory  $T$  and take a sequence  $\Psi := \psi_1, \dots, \psi_n$  of  $T$ -formulas. There is an existential  $T + \text{QFPresburger}$  formula  $\varphi(x_1, \dots, x_n)$  of size polynomial in the size of  $\mathcal{G}$  such that  $f \in P_\Psi(L(\mathcal{G}))$  iff  $\varphi(f(1), \dots, f(n))$  holds.*

In principle there are exponentially many (in  $n$ ) different symbols  $a$  which yield different images  $P_\Psi(a)$ . A naive approach to computing Parikh images would compute the possible images  $f$  of some character  $a$  and calculate their possible sums in words generated by  $\mathcal{G}$ . Since this may require an exponential number of different characters, the naive approach implies at least an exponential running time. We show that this is not the case and that the formula can be polynomial in size.

The main step of the proof is to show that if  $f \in P_\Psi(L(\mathcal{G}))$  then there is a sub-domain  $D_0 \subset D$  of nearly linear (in  $n$  and  $|N|$ ) size such that  $f \in P_\Psi(L(\mathcal{G}) \cap D_0^*)$  (the actual statement is more precise). This essentially reduces the problem to the standard case of a finite alphabet; some details are still needed (like finding the value of the parameters, finding the exact subdomain, etc.), but this is the crucial step. Note that this finitization method does not hold for some models over infinite alphabets, e.g., semilinear data automata of [Figueira and Lin(2022)], which can impose that there are exponentially many different elements of the domain in a given predicate  $\psi$ . A similar result was shown independently to obtain Carathéodory bounds for integer cones [Eisenbrand and Shmonin(2006)] and used to derive parallel results on the complexity of non-emptiness of symbolic tree automata [Raya(2023)].

To show the existence of such a sub-domain, we consider derivations of a word and we focus on the number of times (counts) each production is used. Note that the Parikh image is determined by those counts. Hence, the counts are an “intermediate notion” between the exact derivations and Parikh images and we focus on them. For classic context-free grammars it is well-known that a derivation with given counts exists if and only if those counts satisfy simple arithmetic constraints. Those constraints can be formulated in terms of sums of mappings, which are similar to the Parikh image mappings.

Then, using combinatorial arguments, we show that if we have a large number of such mappings (we use many productions) then there are subsets that have the same effect: i.e. in any derivation we can replace a subset of the productions with another subset without changing the Parikh image of the result. Moreover, we can construct weights for such sets and ensure that each time we make a replacement, the weight drops, which means that the replacing terminates

at some point. The result has to be a derivation with a small number of different productions used.

We illustrate this process with a small example over characters rather than productions. Take the language  $D^*$ . Assume we have three predicates  $\Psi = \psi_1, \psi_2, \psi_3$ . Take a word  $a_1 a_2 a_3 a_4 a_5$  using five different characters and suppose the Parikh image (using vectors to represent the maps) is

$$P_\Psi(a_1 a_2 a_3 a_4 a_5) = (1, 0, 0) + (1, 1, 1) + (1, 1, 0) + (1, 0, 1) + (0, 0, 0) .$$

Notice

$$P_\Psi(a_1 a_2) = (1, 0, 0) + (1, 1, 1) = P_\Psi(a_3 a_4 a_5) = (1, 1, 0) + (1, 0, 1) + (0, 0, 0)$$

and hence we can construct the same Parikh image using only  $a_1$  and  $a_2$ . That is  $P_\Psi(a_1 a_2 a_1 a_2) = P_\Psi(a_1 a_2 a_3 a_4 a_5)$ . Our proof shows that if a large number of different productions are used, the same principle will allow us to find subsets with the same sum.

To make the above intuition formal, we first recall that there is a derivation of a CFG with a given number of times each production is used, if and only if those counts satisfy a simple arithmetical relation [Verma et al.(2005)]; informally: for each nonterminal, the number of times it is introduced and expanded are the same (except for starting nonterminal) plus a condition guaranteeing a variant of being connected; this characterization is similar in spirit and proof to the Euler condition for directed graphs. Formally, for a rule  $A \rightarrow \alpha$  let  $n_{A,\alpha}$  be its count, then

**Lemma 3.3** (cf. [Verma et al.(2005), Thm. 3, 4]). *There is a derivation of a CFG (with starting symbol  $S$ ) that uses  $n_{A,\alpha}$  times a rule  $A \rightarrow \alpha$  if and only if*

$$\begin{aligned} \sum_{(S \rightarrow \alpha) \in P} n_{S,\alpha} &= 1 + \sum_{(A \rightarrow \alpha) \in P} |\alpha|_S \cdot n_{A,\alpha} \\ \forall B \in N \setminus \{S\} \quad \sum_{(B \rightarrow \alpha) \in P} n_{B,\alpha} &= \sum_{(A \rightarrow \alpha) \in P} |\alpha|_B \cdot n_{A,\alpha} \end{aligned}$$

and the underlying graph is connected. Moreover, if  $w$  is generated by such a derivation then

$$|w|_a = \sum_{(a \rightarrow \alpha) \in P} n_{A,\alpha} \cdot |\alpha|_a .$$

Here the underlying graph has  $\mathcal{N}$  as vertices and there is an (undirected) edge  $\{A, B\}$  when  $n_{A,\alpha} > 0$  and  $|\alpha|_B > 0$ , for some  $\alpha$ . In [Verma et al.(2005), Thm. 3, 4] Verma et al. give explicitly a stronger variant of this claim and implicitly formulate Lemma 3.3 in the proof. Moreover, the condition of the underlying graph being connected is also formulated via a Presburger arithmetic formula. While the original construction of such a Presburger formula contained a small error, there are alternative, correct variants in the literature (e.g. [Barner(2006)]); we follow those correct constructions.

We extend this characterization to the case of parametric grammars and reformulate conditions from Lemma 3.3 in terms of mappings, which will allow reasoning on Parikh images and derivations at the same time. We consider an extension of the Parikh image mappings that also assign counts to pairs  $\{s, t\} \times \mathcal{N}$ . A pair  $(s, A)$  indicates how many times the nonterminal  $A$  is the source of a derivation step, and a pair  $(t, A)$  indicates how many times  $A$  is introduced by a derivation step. Thus, we use mappings  $f : \{1, \dots, n\} \cup \{s, t\} \times \mathcal{N} \rightarrow \mathbb{N}$ . Let  $d = n + 2|\mathcal{N}|$



denote the size of the domain of the mapping. The mapping restricted to  $\{1, \dots, n\}$  corresponds to the Parikh image.

Given an instantiation  $\alpha'$  of production  $A \rightarrow \alpha$  we denote by  $f_{A,\alpha,\alpha'}$  the mapping  $f$  with  $f(i) = (P_\Psi(\alpha'))(i)$  for  $i = 1, \dots, n$  (we assume that  $P_\Psi$  ignores the nonterminals), and  $f(s, A) = 1$  and  $f(s, A') = 0$  for  $A \neq A'$  and  $f(t, B)$  is  $|\alpha|_B$ , for each  $B \in \mathcal{N}$ . Note that we associate several mappings with a single production, as there are many instantiations  $\alpha'$  for a fixed  $\alpha$  and on the other hand several instantiations of a rule can have the same mapping.

We consider multisets  $F$  of mappings as above, so  $F : \mathbb{N}^{\{1, \dots, n\} \cup \{s, t\} \times \mathcal{N}} \rightarrow \mathbb{N}$ , which implicitly defines how many times each mapping  $f \in F$  is used; by  $m \cdot \{f\}$  we denote a multiset consisting of  $m$  instances of  $f$ . By  $\sum F$  we denote  $\sum_{f \in F} f$ , which is an element-wise sum:  $(\sum_{f \in F} f)(i) = \sum_{f \in F} f(i)$ . Given a multiset  $F$  of mappings as above we say that a derivation (for some fixed interpretation of parameters) which uses  $n_{A,\alpha,\alpha'}$  times the instantiation  $\alpha'$  of rule  $A \rightarrow \alpha$ , is corresponding to  $F$ , when

$$F = \bigcup_{\substack{(A \rightarrow \alpha) \in P \\ \alpha'}} n_{A,\alpha,\alpha'} \cdot \{f_{A,\alpha,\alpha'}\}.$$

**Lemma 3.4.** *Given a multiset  $F$  of mappings there is a derivation corresponding to it if and only if*

$$\left(\sum F\right)(s, S) = 1 + \left(\sum F\right)(t, S) \tag{1a}$$

$$\left(\sum F\right)(s, A) = \left(\sum F\right)(t, A) \quad \text{for all } B \in N \setminus \{S\} \tag{1b}$$

and the underlying graph is connected. Moreover, this derivation yields a word with Parikh image  $(\sum F)$  restricted to  $\{1, \dots, n\}$ .

Here the underlying graph has nodes  $\{A : (\sum F)(s, A) > 0\}$  and edges  $\{A, B\}$  when there is  $f \in F$  such that  $f(s, A), f(t, B) > 0$ .

*Proof.* After fixing the parameters and the possible instantiations of the rules, the parametric CFG becomes a CFG grammar over a finite-size alphabet. We show that condition (1) from the Lemma is equivalent to the one from Lemma 3.3. Note that the condition that the underlying graph is connected is the same in proven Lemma and in Lemma 3.3.

Suppose that there is a derivation corresponding to  $F$ . Denote by  $n_{A,\alpha,\alpha'}$  the counts of the instantiations  $\alpha'$  of the rules  $A \rightarrow \alpha$ . Then we can treat the parametric grammar as an ordinary CFG, with a pair  $(A \rightarrow \alpha), \alpha'$  being treated as a single rule  $A \rightarrow \alpha'$ . Hence the numbers  $n_{A,\alpha,\alpha'}$  satisfy the following equations as in Lemma 3.3:

$$\begin{aligned} \sum_{\substack{(S \rightarrow \alpha) \in P \\ \alpha'}} n_{S,\alpha,\alpha'} &= 1 + \sum_{\substack{(A \rightarrow \alpha) \in P \\ \alpha'}} |\alpha|_S \cdot n_{A,\alpha,\alpha'} \\ \forall B \in N \setminus \{S\} \quad \sum_{\substack{(B \rightarrow \alpha) \in P \\ \alpha'}} n_{B,\alpha,\alpha'} &= \sum_{\substack{(A \rightarrow \alpha) \in P \\ \alpha'}} |\alpha|_B \cdot n_{A,\alpha,\alpha'} \end{aligned}$$

Observe that  $\sum_{(A \rightarrow \alpha) \in P, \alpha'} n_{S,\alpha,\alpha'}$  is  $(\sum F)(s, A)$  while  $\sum_{(A \rightarrow \alpha) \in P, \alpha'} |\alpha|_S \cdot n_{A,\alpha,\alpha'}$  is  $(\sum F)(t, A)$ . Hence the two equations are a reformulation of the conditions (1).

In the other direction, the argument is similar, but note that for a mapping  $f \in F$  with count  $n_f$  we need to give counts for each production  $A \rightarrow \alpha$  and its instantiation  $\alpha'$  such that  $f_{A,\alpha,\alpha'} = f$  and that the sum of those counts is  $n_f$ . This is done arbitrarily: given a mapping  $f \in F$  with count  $n_f$  we choose arbitrarily a single such rule  $A \rightarrow \alpha$  and its instantiation  $\alpha'$  such that  $f_{A,\alpha,\alpha'} = f$  and set  $n_{A,\alpha,\alpha'}$  to  $n_f$ . The same argument as above shows that the equations from the Lemma are just reformulations of equations from Lemma 3.3.

The claim on the Parikh image follows: the derivation uses an instantiation  $\alpha'$  of the  $A \rightarrow \alpha$  rule  $n_{A,\alpha,\alpha'}$  times, which yields that the contribution of the letters in  $\alpha'$  is as stated.  $\square$

Now the idea is that if  $F$  corresponds to a derivation, then there is a different multiset  $F'$  with the same sum  $\sum F = \sum F'$ . Moreover,  $F'$  satisfies conditions (1) but uses fewer different mappings than  $F$ . In particular,  $F'$  also corresponds to some derivation.

**Lemma 3.5.** *Let a multiset of mappings  $F$  satisfy condition (1) from Lemma 3.4. Suppose that there are two multisets  $F', F'' \subseteq F$  such that  $\sum F' = \sum F''$ ,  $\sum (F \setminus F') \cup F'' = \sum F$  and the underlying graph for  $(F \setminus F') \cup F''$  is connected. Then  $(F \setminus F') \cup F''$  satisfies condition (1) from Lemma 3.4.*

*Proof.* As  $F' \subseteq F$  the  $F \setminus F'$  is well defined and  $\sum ((F \setminus F') \cup F'') = (\sum F) - (\sum F') + (\sum F'')$ . By assumption  $\sum F' = \sum F''$  we have that  $\sum ((F \setminus F') \cup F'') = \sum F$  and hence the assumptions of Lemma 3.4 are met for  $(F \setminus F') \cup F''$ .  $\square$

We now show that given a large enough set of mappings we can always find two its subsets of the same sum. Note that here we do not use multisets, but rather simply sets.

**Lemma 3.6.** *Given a set of  $k$  different mappings  $f : \{0, 1, \dots, d\} \rightarrow \{0, 1, \dots, \ell\}$ , where  $d > 3$  and  $k \geq 2d \log(d\ell)$ , there are two disjoint subsets of this set that have the same sum.*

*Proof.* It is enough to show that there are two subsets of the same sum, they can be made disjoint by removing their intersection.

There are  $2^k$  different subsets and at the same time each value of the sum of mappings is between 0 and  $k\ell$ , so in total there are at most  $(k\ell + 1)^d$  different possible sums and so it is enough to show that  $(k\ell + 1)^d < 2^k$ . That is  $d \log(k\ell + 1) < k$ , i.e.  $d < \frac{k}{\log(k\ell + 1)}$ . As  $\frac{k}{\log(k\ell + 1)}$  is increasing for  $k \geq 2$ , it is enough to verify for  $k = 2d \log(d\ell)$  that :

$$d < \frac{2d \log(d\ell)}{\log(2d \log(d\ell) + 1)} = d \frac{\log((d\ell)^2)}{\log(2d \log(d\ell) + 1)} .$$

Which is equivalent to (setting  $x = d\ell$ )

$$\log(x^2) > \log(2x \log x + 1)$$

As  $\log$  is an increasing function, then it is enough to show that  $x^2 > 2x \log x + 1$ , which clearly holds for  $x > 3$ , so in particular for  $d > 3$ .  $\square$

**Lemma 3.7.** *Given a multiset  $F$  of mappings such that there is a derivation corresponding to it, there is a multiset  $F'$  of mappings corresponding to it,  $\sum F = \sum F'$  and  $F'$  contains at most  $|\mathcal{N}| + 2d \log d\ell$  different mappings, where  $\ell$  is an upper-bound on the value of each mapping in  $F$ .*

*Proof.* The idea is as follows: given a multiset  $F$  we use Lemma 3.6 to claim that  $F$  uses few different mappings or to find  $F_1, F_2 \subset F$  such that both  $(F \setminus F_2) \cup F_2$  and  $(F \setminus F_2) \cup F_1$  correspond

to a valid derivation and  $\sum(F \setminus F_1) \cup F_2 = \sum F = \sum(F \setminus F_2) \cup F_1$ ; we then replace  $F$  with one of them. To guarantee that the replacement terminates, we introduce a natural well-founded order on multisets of mappings and show that (at least) one of  $(F \setminus F_1) \cup F_2$  and  $(F \setminus F_2) \cup F_1$  is strictly smaller in this order than  $F$ . This shows that the replacement terminates at some point and so the resulting multiset uses few different mappings, which shows the claim of the Lemma.

Consider the *set* of mappings in  $F$ , let us linearly order them in some arbitrary way as  $f_1, f_2, \dots, f_m$ . We now treat the multisets of mappings as mappings themselves, i.e.  $F'(f_i)$  gives the number of times  $f_i$  is in  $F'$ . We introduce a linear well-founded order  $\leq$  on the multisets of mappings: we first compare by the number of non-zero components (a smaller number implies the image is smaller according to  $\leq$ ), i.e. if  $|\{i : F'(f_i) > 0\}| < |\{i : F''(f_i) > 0\}|$  then  $F' < F''$  and otherwise we compare the mappings lexicographically by coordinates, i.e. if  $|\{i : F'(f_i) > 0\}| = |\{i : F''(f_i) > 0\}|$  and  $i$  satisfies  $F'(f_j) = F''(f_j)$  for  $j < i$  and  $F'(f_i) < F''(f_i)$  then  $F' < F''$ . By standard arguments, this is a well-founded order (as the order on  $\mathbb{N}$  is well-founded and a lexicographic order such that the order on each component is well-founded is itself well-founded).

We arbitrarily choose  $F_0 \subseteq F$  mappings from  $F$ , where  $|F_0| \leq |\mathcal{N}|$ , which guarantee that the underlying graph is connected; this can be done as the underlying graph is connected, by Lemma 3.3, and has  $|\mathcal{N}|$  many vertices. The  $F_0$  will be added to the final multiset of mapping to guarantee that the underlying graph is connected. Let  $F' = F \setminus F_0$  be the remaining multiset of mappings. If  $F'$  has more than  $2d \log d\ell$  different mappings, then by Lemma 3.6 there are two disjoint sets  $F_1, F_2 \subset F'$  such that  $\sum F_1 = \sum F_2$ . Let  $F'_1 = (F' \setminus F_1) \cup F_2$  and  $F'_2 = (F' \setminus F_2) \cup F_1$ , where  $F'_1$  and  $F'_2$  are multisets. By the same Lemma,  $\sum F'_1 = \sum F'_2 = \sum F'$ . Note also that  $F'_1, F'_2$  cannot have more non-zero components than  $F'$ , as we are only adding mappings which are already in  $F'$ . Hence, if  $F'_1$  has a non-zero component, it is also non-zero in  $F'$ , and similarly for  $F'_2$ . Hence  $F'$  cannot be smaller than  $F'_1$  or  $F'_2$  because it has less non-zero components.

As  $F_1 \neq F_2$ , consider the smallest  $i$  such that  $F_1(f_i) \neq F_2(f_i)$ . If  $F_1(f_i) > F_2(f_i)$  then  $F'_1(f_i) < F'(f_i)$  and  $F'_1(f_j) = F'(f_j)$  for  $j < i$ ; hence  $F'_1 < F'$  and we replace  $F'$  with  $F'_1$  (note that it could be that  $F'_1$  has fewer non-zero components than  $F'$ , in which case the above calculations are superfluous).

If  $F_1(f_i) < F_2(f_i)$  then similarly  $F'_2 < F'$ .

We can proceed in this manner as long as there are at least  $2d \log d\ell$  different mappings in  $F'$ . Since  $\leq$  is well-founded, the process terminates. Let  $F''$  be the final set; it uses less than  $2d \log(d\ell) = 2(n + 2|\mathcal{N}|) \log((n + 2|\mathcal{N}|)\ell)$  different mappings. Then  $F_0 \cup F''$ , where  $F_0$  are the initially chosen  $|\mathcal{N}|$  many mappings that guarantee connectedness, we get the desired multiset: the underlying graph is connected thanks to  $F_0$  and  $\sum(F_0 \cup F'') = \sum F$  by easy induction and then condition (1) from Lemma 3.4 holds and so by this Lemma there is a derivation corresponding to  $F_0 \cup F''$ .  $\square$

This is enough to give a proof of Theorem 3.2: we can construct a formula for theory  $T$  combined with quantifier-free Presburger arithmetic representing the conditions needed. We then finally existentially quantify all intermediate variables to leave the free variables  $x_1, \dots, x_n$  counting the number of times each  $\psi_i$  is satisfied.

The formula first guesses the values  $X$  of the parameters  $\mathcal{X}$  for the grammar. Then it guesses  $|\mathcal{N}| + 2d \log d\ell$  mappings  $\{0, 1, \dots, n\} \cup \{s, t\} \times \mathcal{N} \rightarrow \{0, 1, \dots, \ell\}$ , where  $\ell$  is the maximal length of  $\mathcal{G}$  rules (we can represent a mapping with  $n + 1 + 2|\mathcal{N}|$  integer variables ranging over  $\{0, 1, \dots, \ell\}$ ). It also guesses the counts of each mapping, yielding a multiset  $F$ . For each of those mappings it guesses the rule of the grammar and its instantiation. It verifies the correctness of those guesses:

1. That  $\sum F$  satisfies (1).
2. The underlying graph is connected
3. That each  $f \in F$  indeed corresponds to a guessed rule  $A \rightarrow \alpha$  and an instantiation  $\alpha'$ .

If all tests are satisfied then the formula accepts, otherwise it rejects. Notice that the first condition is a quantifier-free Presburger condition. For the second condition, for each  $f$ , we need to guess an assignment to  $\mathcal{V}$ . (We can use a fresh copy of  $\mathcal{V}$  for each of the  $|N| + 2d \log d\ell$  mappings). Via a disjunction over all grammar rules, the formula guesses the corresponding rule, and verifies that the number of symbols in  $\alpha$  satisfying  $\psi_i$  matches  $f(i)$  and similarly for the nonterminal symbols (on both sides). To see how to do this with a polynomial-size formula, let  $\alpha_1 \cdots \alpha_m$  be the right-hand side of the guessed rule. For each  $\psi_i$ , introduce a 0-1 variable  $c_j^i$  for each  $\alpha_j$ . This variable is 0 if  $\psi_i(\alpha_j)$  does not hold and 1 otherwise. Then we check  $f(i) = \sum_j c_j^i$ . For this, we only require Boolean combinations of the two theories.

Each free-variable  $x_i$  of the formula takes the value  $\sum_{f \in F} f(i)$  for all  $1 \leq i \leq n$ .

If there is a satisfying assignment to the formula, giving a mapping  $f$ , then indeed there is  $w \in L_X(\mathcal{G})$  such that  $P_\Psi(w)$  equals  $f$  on the first  $n$  components. We guessed the value of the parameters and  $F$  corresponds to a derivation, by Lemma 3.4 and the Parikh image of the derived word is indeed  $f = \sum F$  restricted to components  $1, \dots, n$ .

In the other direction, if there is  $w \in L(\mathcal{G})$  then by definition for some  $X$  we have  $w \in L_X(\mathcal{G})$ . Fix a derivation of  $w$  and let  $F$  be the corresponding set of mappings;  $F$  satisfies the conditions from Lemma 3.4, so in particular satisfies 2. Then by Lemma 3.7 we can assume without loss of generality that there are at most  $|N| + 2d \log d\ell$  different mappings in  $F$ . From this we can construct assignments to the variables representing  $F$  and its counts that satisfy condition 1. Since each  $f \in F$  corresponds to a rule in a concrete derivation, it also follows that condition (3) is also satisfied.

### 3.3 Complexity

To measure the complexity of Parikh images of parametric grammars, we consider the problem of deciding whether  $P_\Psi(L(\mathcal{G})) \cap \llbracket \Phi \rrbracket = \emptyset$ , for a given existential Presburger formula  $\Phi$  over variables  $x_1, \dots, x_n$ . Here,  $\llbracket \Phi \rrbracket$  refers to the set of assignments  $f : \{1, \dots, n\} \rightarrow \mathbb{N}$  such that  $f(1), \dots, f(n)$  satisfies  $F$ . Because we are able to compute a polynomial representation of the Parikh image, the complexity remains similar to  $T$ .

**Theorem 3.8.** *Let  $T$  be solvable in complexity class  $\mathcal{C}$ . Then the complexity of Parikh over images of parametric grammars  $T$  is in  $\exists\mathcal{C}$ . In particular, if  $T$  is solvable in P, NP, PSPACE then the Parikh images of parametric grammars are in NP, NP, PSPACE, respectively.*

First, we construct  $\varphi(x_1, \dots, x_n)$  as in Theorem 3.2. We then need to find an assignment to  $x_1, \dots, x_n$  such that  $\varphi(x_1, \dots, x_n) \wedge \Phi(x_1, \dots, x_n)$  holds.

If there is  $w \in P_\Psi(L(\mathcal{G})) \cap \llbracket \Phi \rrbracket$ , fix a derivation of  $w$  and let  $F$  be the corresponding set of mappings;  $F$  satisfies (1) and  $\Phi$ . For each different  $f \in F$  consider its count  $n_f$  in  $F$ . Consider conditions 1 and 3 and  $\Phi$ . We claim that if they are satisfied, then they are satisfied for counts  $\{n_f\}_{f \in F}$  that are at most exponential. Given the set of different mappings in  $F$ , condition (1) and  $\Phi$  can be interpreted as a system of linear equations on  $\{n_f\}_{f \in F}$ , and condition (3) requires a guess of the rule and its instantiation, which can be done separately. The check whether the underlying graph is connected can also be done separately, as it does not depend on  $\{n_f\}_{f \in F}$ .

Then the system of equations is over  $|N| + 2d \log d\ell$  variables and with polynomial-size constants. By standard results, if it has a solution, it has one that is at most exponential-size.

Thus, the integer variables have solutions encodable via a polynomial number of bits. If  $T$  is solvable in P or NP, then we can decide  $P_\Psi(L(\mathcal{G})) \cap \llbracket \Phi \rrbracket \neq \emptyset$  in NP. If  $T$  is PSPACE, then the problem is also PSPACE.

## 4 Abstraction of String Constraints

In this section we outline an application of the symbolic Parikh image abstraction to solving string constraints. String constraints arise naturally during symbolic execution analysis of programs, where the string data type is ubiquitous. During symbolic execution, the potential paths of the program are explored and the constraints on the variables are collected. For example, the positive branch of an if-statement with the condition  $x = ab$  will result in the condition  $x = ab$  being added to the collection of constraints on the path. If an error state is discovered, a check whether the collection of constraints on the path to the state are satisfiable is made. That is, is there some assignment to the variables that would cause this path to be executed? Because many of these checks will be made during analysis, it is important that the string constraint solver is efficient.

In this setting, we consider constraints that may contain string- and integer-valued variables, Presburger arithmetic, and a number of common string operations such as concatenation and containment in a regular language.

To help improve solver efficiency, we may consider using the Parikh image abstraction to help identify unsatisfiable constraints and avoid a potentially costly proof search by the solver. The goal of the abstraction is to overapproximate the satisfiable instances. This means an unsatisfiable abstracted instance implies the original instance was also unsatisfiable. That is, we allow false positives, but not false negatives. We may hope that the abstracted constraint – which no longer includes string variables but integer variables representing the Parikh image of the strings – can be solved more quickly than the unabstracted constraint.

In the next sections, we introduce the constraint language, describe our abstraction of the input constraints, then describe a modified representation of the Parikh image of a regular expression.

### 4.1 The Constraint Language

We focus on a subset of the QF\_SLIA theory of SMT-LIB 2.6 [Barrett et al.(2017)]. That is, string constraints with linear integer arithmetic. The constraint language is explained below. Because we focus on SMT-LIB, our constraint language only has constraints that a string is in a regular language. Of course, we can easily extend this language to support containment checks in the language of a parametric context-free grammar.

Note, we assume all formulas are given in negation normal form. This is because, as explained below, abstracting negative equations can lead to false negatives. Our constraint language contains the following components.

- String-valued expressions

$$s, s_1, s_2 := w \mid x \mid str.++(s_1, s_2) \mid str.replace(s, s_1, s_2) \mid str.substr(s, i_1, i_2) .$$

That is, string-valued expressions can be string literals  $w$ , string variables  $x$ , the concatenation  $str.++$  of two string-valued expressions  $s_1$  and  $s_2$ , the result of replacing the first occurrence of  $s_1$  in  $s$  with  $s_2$ , or the substring of  $s$  from position  $i_1$  of length  $i_2$ , for integer expressions  $i_1$  and  $i_2$ .

- Boolean-valued string expressions

$$\begin{aligned} & \text{str.in\_re}(s, r) \mid \neg \text{str.in\_re}(s, r) \mid s_1 = s_2 \mid \\ & \text{str.contains}(s_1, s_2) \mid \text{str.prefixof}(s_1, s_2) \mid \text{str.suffixof}(s_1, s_2) . \end{aligned}$$

That is, the Boolean-valued expressions can be the test that  $s$  is (or is not) in the regular expression  $r$ , that  $s_1$  equals  $s_2$ ,  $s_1$  contains the contiguous substring  $s_2$ ,  $s_1$  is a prefix of  $s_2$ , or  $s_1$  is a suffix of  $s_2$ . We support the regular expressions supported by Z3 which are detailed below. Boolean expressions may appear in any positive Boolean combination.

- The integer-valued expressions

$$\text{str.len}(s)$$

for string expression  $s$ . Note, other integer valued expressions (that do not include strings) that are supported by Z3 are also permitted. In particular, quantifier-free Presburger arithmetic (or linear integer arithmetic).

- Regular expressions have a standard interpretation and are

$$r, r_1, r_2 := a \mid \text{re.in\_range}(a, a') \mid r_1 r_2 \mid r_1 \vee r_2 \mid r_1 \wedge r_2 \mid \neg r \mid r^* \mid r^+ \mid r^? \mid r^{l,h} \mid \emptyset \mid \Sigma$$

where  $a$  is a concrete character;  $\text{re.in\_range}(a, a')$  is any character between  $a$  and  $a'$  (inclusive);  $r_1 r_2$  is concatenation;  $r_1 \vee r_2$ ,  $r_1 \wedge r_2$ , and  $\neg r$  are Boolean operations;  $r^*$  is 0 or more consecutive matches of  $r$ ;  $r^+$  is 1 or more consecutive matches of  $r$ ;  $r^?$  is 0 or 1 matches of  $r$ ;  $r^{l,h}$  is between  $l$  and  $h$  matches of  $r$  where  $l$  is a nonnegative integer and  $h$  is a nonnegative integer or infinity;  $\emptyset$  matches no word; and  $\Sigma$  matches any character.

## 4.2 Abstraction of Input

Let  $\Psi := \psi_1, \dots, \psi_n$  be the predicates of the Parikh image. We assume that  $\psi_1 = \top$  as it is convenient for encoding the length of a string. We will take a vector view of the Parikh image mappings  $f$ . That is, we will denote  $f$  as a vector  $(f_1, \dots, f_n)$  where  $f_i = f(i)$ . This means we can refer to a vector of variables or expressions that represent a Parikh image.

Let  $\varphi_\top$  be the Parikh image of the regular expression  $\Sigma^*$  that matches any string. We use  $\varphi_\top$  to assert that any vector of expressions  $\vec{c} = (c_1, \dots, c_n)$  encodes the Parikh image of a string. E.g., for predicates  $\top, \perp$ , the counts 0, 1 are not a valid Parikh image and  $\varphi_\top(0, 1)$  does not hold.

Our approach creates an overapproximation abstraction of the input SMT-LIB formula. Each string expression  $s$  is abstracted as a vector  $\vec{c}^s$  where each component  $c_i^s$  is an expression counting the number of times a character satisfies  $\psi_i$  in the value of  $s$ . Similarly, regular expressions are abstracted as a formula  $\varphi$  recognising the Parikh image of the expression.

This means all string variables  $x$  are abstracted with a vector  $\vec{x}^\Psi = (x_1^\Psi, \dots, x_n^\Psi)$ . The variable  $x_i^\Psi$  counts the number of occurrences of characters matching the predicate  $\psi_i$  in the value assigned to  $x$ . We assert  $\varphi_\top(\vec{x}^\Psi)$  for each abstracted string variable.

Each string expression  $s$  is abstracted recursively. The translation may introduce new variables, for which additional side conditions need to be asserted. The side conditions are collected as a side-effect of the translation. Pseudo-code is given in Algorithm 1 and explained below. The Assert function adds the assertion to the output formula.

The abstraction of  $w$  directly counts the number of characters of  $w$  that satisfy each predicate. Since  $w$  is a concrete string, this is a straightforward character-by-character check against each predicate. For a string variable  $x$  we use the abstraction variables  $\vec{x}^\Psi$ .



The *str.replace* operation is the most subtle and reflects the semantics of *str.replace* in SMT-LIB. We introduce fresh variables  $\bar{y}^\Psi$  to store the abstracted result of the replace. There are three possible outcomes. If  $s_2$  does not appear in  $s_1$ , then  $s_1$  is unchanged and we assert  $\bar{y}^\Psi = \bar{c}^1$  where  $\bar{c}^1$  is the abstraction of  $s_1$ . If  $s_2$  appears in  $s_1$ , then the first instance of  $s_2$  is removed from  $s_1$  and replaced with  $s_3$ . The final case is when  $s_2$  is the empty string. In this case, the SMT-LIB semantics is that  $s_3$  is prepended onto  $s_1$ . This is encoded by  $\bar{y}^\Psi = \bar{c}^1 + \bar{c}^3$ . Notice, since the values of  $\bar{y}^\Psi$  are derived from  $\bar{c}^1$ ,  $\bar{c}^2$ , and  $\bar{c}^3$ , we do not need to assert  $\varphi_\top$  as they are already implied.

Finally, for *str.substr* we again introduce fresh variables  $\bar{y}^\Psi$ . We ignore the integer arguments  $i_1$  and  $i_2$  and simply require that  $\bar{y}^\Psi$  is (point-wise) contained within  $\bar{c}^1$ . We assert  $\varphi_\top$  to ensure the values represent a string.

---

**Algorithm 1:** AbstractSEXP( $s$ )

---

```

1 if  $s = w$  then
2   | return  $\bar{c}$  where each  $c_i$  is the count of characters satisfying  $\psi_i$ 
3 if  $s = x$  then
4   | return  $\bar{x}^\Psi$ 
5 if  $s = \text{str.}++(s_1, s_2)$  then
6   | return AbstractSEXP( $s_1$ ) + AbstractSEXP( $s_2$ )
7 if  $s = \text{str.replace}(s_1, s_2, s_3)$  then
8   |  $\bar{c}^1 \leftarrow \text{AbstractSEXP}(s_1)$ ;  $\bar{c}^2 \leftarrow \text{AbstractSEXP}(s_2)$ ;  $\bar{c}^3 \leftarrow \text{AbstractSEXP}(s_3)$ ;
9   | Let  $\bar{y}^\Psi$  be fresh integer variables;
10  | Assert( $\bar{y}^\Psi = \bar{c}^1 \vee \bar{y}^\Psi = \bar{c}^1 - \bar{c}^2 + \bar{c}^3 \vee (\bar{c}^2 = 0 \wedge \bar{y}^\Psi = \bar{c}^1 + \bar{c}^3)$ );
11  | return  $\bar{y}^\Psi$ 
12 if  $s = \text{str.substr}(s_1, i_1, i_2)$  then
13  |  $\bar{c}^1 \leftarrow \text{AbstractSEXP}(s_1)$ ;
14  | Let  $\bar{y}^\Psi$  be fresh integer variables;
15  | Assert( $\bar{y}^\Psi \leq \bar{c}^1$ ); Assert( $\varphi_\top(\bar{y}^\Psi)$ );
16  | return  $\bar{y}^\Psi$ 

```

---

We can then abstract Boolean expressions contained in the input. We first convert each assertion to negation normal form. This is because we can abstract, for example,  $x_1 = x_2$  but not  $x_1 \neq x_2$ .<sup>2</sup> We then substitute maximal subexpressions according to the following scheme.

- $\neg \text{str.in\_re}(s, r)$  is replaced by  $\varphi_{\neg r}(\text{AbstractSEXP}(s))$  where  $\varphi_{\neg r}$  is the Parikh image of the complement of the language accepted by  $r$  (discussed in the next section).
- $\text{str.in\_re}(s, r)$  is replaced by  $\varphi_r(\text{AbstractSEXP}(s))$  where  $\varphi_r$  is the Parikh image of the language accepted by  $r$ .
- $s_1 = s_2$  is replaced by  $\text{AbstractSEXP}(s_1) = \text{AbstractSEXP}(s_2)$ .
- $\text{str.contains}(s_1, s_2)$  is replaced by  $\text{AbstractSEXP}(s_1) \geq \text{AbstractSEXP}(s_2)$ .
- $\text{str.prefixof}(s_1, s_2)$  and  $\text{str.suffixof}(s_1, s_2)$  are replaced by  $\text{AbstractSEXP}(s_1) \leq \text{AbstractSEXP}(s_2)$ .

---

<sup>2</sup>Because the only satisfying assignments to  $x_1$  and  $x_2$  may happen to have the same Parikh image. In this case we cannot assert that the Parikh image of  $x_1$  is not equal to the Parikh image of  $x_2$ , making a satisfiable formula unsatisfiable.

- $str.len(s)$  is replaced by  $c_1$  where  $\vec{c}$  is the result of  $\text{AbstractSEXP}(s)$ . Recall  $\psi_1$  was assumed to be  $\top$ , so  $c_1$  gives the length of the string.

### 4.3 Construction of the Parikh Image

The abstraction above uses  $\varphi_r$ , which is the symbolic Parikh image abstraction of the regular expression  $r$ . We describe how we encode  $\varphi_r$  as an SMT-LIB formula. Our encoding goes via symbolic automata and is slightly different to the proof of Theorem 3.8. The alternative encoding is driven by the concrete transitions of the automaton representing  $r$ , rather than a nondeterministically chosen subset of the states and transitions. This is because when transitions are represented by variables, many clauses need to contain a disjunction over all possible instantiations of the variables, causing an undesirable polynomial blow-up. This new encoding may require  $2sn \log(n)$  character variables, where  $s$  is the number of transitions of the symbolic automaton. This is theoretically worse than the  $2(n + 2|Q|) \log(n + 2|Q|)$  characters needed in the proof of Theorem 3.8. In the next section we describe some mitigating optimisations.

Given a regex  $r$ , we build an equivalent symbolic automaton  $\mathcal{A}$ . We briefly recall the definition we use of a symbolic automaton. It is equivalent to parametric context-free grammars where all productions are of the form  $(A, \alpha, \varphi(curr))$  (i.e. no parameters) and  $\alpha = curr B$  or  $\alpha = \varepsilon$ .

For a given theory  $T$ , a symbolic automaton is a tuple  $(Q, \Delta, q_0, F)$  where  $Q$  is a finite set of states,  $\Delta \subseteq Q \times T(curr) \times Q$  is the transition relation,  $q_0$  is the initial state, and  $F \subseteq Q$  is the set of final states. A run over a word  $a_1 \dots a_\ell$  is a sequence of transitions  $(q_0, \varphi_1, q_1)(q_1, \varphi_2, q_2) \dots (q_{\ell-1}, \varphi_\ell, q_\ell)$  where  $T \models \varphi_i(a_i)$  for all  $i$  and  $q_\ell \in F$ .

Using predicates that are Boolean combinations<sup>3</sup> of  $\varphi(curr) := curr = a$  and of  $\varphi(curr) := str.in\_re(curr, re.in\_range(a, a'))$ , standard constructions can be used to obtain a symbolic automaton that is equivalent to a regular expression as defined above. From  $\mathcal{A}$  we build a formula  $\varphi(c_1, \dots, c_n)$ , where the free variable  $c_i$  indicates the number of characters satisfying predicate  $\psi_i$ .

Let  $\text{Labels}(\mathcal{A})$  be the set of predicates appearing on the transitions of  $\mathcal{A}$ . We use  $p$  to denote these predicates to avoid confusion with the output predicates  $\psi_1, \dots, \psi_n$ . For a predicate  $p \in \text{Labels}(\mathcal{A})$ , let  $\text{TransLabelled}(p)$  be the set of transitions of  $\mathcal{A}$  labelled by the predicate  $p$ . Let  $\text{Labels}(\mathcal{A})$  be  $p_1, \dots, p_m$ . Using a corrected version of Verma et al. [Verma et al.(2005), Barner(2006)], we can build a linear-sized existential Presburger formula from  $\mathcal{A}$  with free variables  $c_{p_1}, \dots, c_{p_m}$  where  $c_p$  indicates how many transitions labelled  $p$  appear in a run of  $\mathcal{A}$ . That is, given a run  $\rho = t_1 \dots t_\ell$  of  $\mathcal{A}$  and a transition  $t$ , let  $|\rho|_t$  be the number of occurrences of  $t$  in  $\rho$ . Then, for a predicate  $p$ , let  $|\rho|_p = \sum_{t \in \text{TransLabelled}(p)} |\rho|_t$ . If  $\varphi(c_{p_1}, \dots, c_{p_m})$  holds, then there is a run  $\rho$  of  $\mathcal{A}$  with  $|\rho|_{p_i} = c_{p_i}$  for all  $i$ .

We split the target counts  $c_1, \dots, c_n$  of the output predicates  $\psi_1, \dots, \psi_n$  between the counts contributed by each of the transition labels in  $\text{Labels}(\mathcal{A})$ . To do this we introduce variables  $c_i^p$  indicating that the predicate  $p$  accounts for  $c_i^p$  characters satisfying  $\psi_i$ . That is for all  $i$

$$c_i = \sum_{p \in \text{Labels}(\mathcal{A})} c_i^p.$$

We then check, for each  $p \in \text{Labels}(\mathcal{A})$  whether there exists a sequence of characters  $a_1, \dots, a_{\ell'}$  such that  $\ell' = c_p$  (recall  $c_p$  is the number of times  $p$  appeared as the label of a

<sup>3</sup>Boolean combinations may arise during, e.g., product constructions and complementation.

transition in the run). Moreover, for each output predicate  $\psi_i$ , we require that  $c_i^p$  is the number of characters  $a$  in  $a_1, \dots, a_{\ell'}$  such that  $\psi_i(a)$  holds.

By Lemma 3.6, the sequence  $a_1, \dots, a_{\ell'}$  needs at most  $2n \log(n)$  different characters. We introduce character variables  $a_i^p$  for  $1 \leq i \leq 2n \log(n)$  and character count variables  $k_j^p$ . That is, character  $a_i^p$  appears  $k_j^p$  times in  $a_1, \dots, a_{\ell'}$ . Additionally, we naturally require that  $a_i$  satisfies  $p$ .

All together, we assert (using  $\varphi$ ) that

- the counts for each label ( $c_p$ ) is in the Parikh image of the automaton (using  $\varphi$ ),
- the total count for each output predicate ( $c_i$ ) is the sum of counts of the labels satisfying the predicates (using  $\varphi_{\text{sum}}$ ), and
- for each label, we assert (using  $\varphi_{\text{labels}}$ ) that
  - the count for that label is spread across  $2n \log(n)$  characters (using  $\varphi_{\text{lcounts}}$ , recalling  $a_i^p$  appears  $k_j^p$  times),
  - each character satisfies the label (using  $\varphi_{\text{preds}}$ ), and
  - the counts of the labels satisfying a predicate is correct (using  $\varphi_{\text{pcounts}}$ ).

That is, recalling  $\text{Labels}(\mathcal{A}) = p_1, \dots, p_m$ , we define  $\varphi_r(c_1, \dots, c_n)$  to first use the non-symbolic Parikh image  $\varphi(c_{p_1}, \dots, c_{p_m})$  to calculate how many time each transition label can occur. Then it uses  $\varphi_{\text{sum}}$  to assert that the total number of times  $\psi_i$  is satisfied is distributed across each transition label. Finally  $\varphi_{\text{labels}}$  bridges between the number of labels, the number of characters satisfying those labels, and the number of times each output predicate  $\psi_i$  is satisfied. That is,  $\varphi_r(c_1, \dots, c_n) :=$

$$\exists_{p \in \text{Labels}(\mathcal{A})} c_p, c_i^p, a_j^p, k_j^p \cdot \varphi(c_{p_1}, \dots, c_{p_m}) \wedge \varphi_{\text{labels}} \wedge \varphi_{\text{sum}}$$

$\begin{matrix} 1 \leq i \leq n \\ 1 \leq j \leq m \end{matrix}$

where

$$\begin{aligned} \varphi_{\text{sum}} &:= \bigwedge_{1 \leq i \leq n} \left( c_i = \sum_{p \in \text{Labels}(\mathcal{A})} c_i^p \right) && \left[ \begin{array}{l} \text{output counts are split between the} \\ \text{transition labels} \end{array} \right] \\ \varphi_{\text{labels}} &:= \bigwedge_{p \in \text{Labels}(\mathcal{A})} (\varphi_{\text{lcounts}} \wedge \varphi_{\text{preds}} \wedge \varphi_{\text{pcounts}}) && \left[ \begin{array}{l} \text{correctness of the labels, see be-} \\ \text{low} \end{array} \right] \\ \varphi_{\text{lcounts}} &:= c_p = \sum_{1 \leq j \leq 2n \log(n)} k_j^p && \left[ \begin{array}{l} \text{the count of transitions labelled by} \\ p \text{ is split between } 2n \log(n) \text{ different} \\ \text{characters, each appearing } k_j^p \text{ times} \end{array} \right] \\ \varphi_{\text{preds}} &:= \bigwedge_{1 \leq j \leq 2n \log(n)} p(a_j^p) && \left[ \begin{array}{l} \text{each of the } 2n \log(n) \text{ characters sat-} \\ \text{isfies } p \end{array} \right] \\ \varphi_{\text{pcounts}} &:= \bigwedge_{1 \leq i \leq n} \left( c_i^p = \sum_{1 \leq j \leq 2n \log(n)} k_j^p \times \psi_i^{0,1}(a_j^p) \right) && \left[ \begin{array}{l} \text{the number of times } \psi_i \text{ is satisfied} \\ \text{by transitions labelled } p \text{ is the sum} \\ \text{of the } 2n \log(n) \text{ character counts} \\ \text{that satisfy } \psi_i \end{array} \right] \end{aligned}$$

and  $\psi_i^{0,1}(a)$  is 1 when  $a$  satisfies  $\psi_i$  and 0 otherwise.

## 4.4 The Overapproximation

We put everything together to gain an overapproximation of an SMT-LIB formula containing string and integer expressions. We abstract each string variable and expression as a sequence of integer variables – one for each output predicate. We replace string expressions with their abstracted equivalent, which may include Parikh images of symbolic automata. If the abstracted constraint is unsatisfiable, we conclude that the original constraint was also unsatisfiable, and avoid reasoning over the string data type. We describe our experiments in the next section.

## 4.5 Extensions

Here we discuss possible extensions of our constraint language and technique. First, we could additionally allow context-free constraints, not just regular expressions. This is allowed already by some string solvers (e.g. TRAU [Abdulla et al.(2018), Abdulla et al.(2017)]), but this is not yet supported by SMT-LIB. The technique in this section easily extends to context-free constraints since our general results in Section 3 concerns symbolic context-free grammars. This can also be extended to symbolic pushdown automata with the restriction that the number of push symbols in a transition is small. Second, using parametric grammars, we could support a currently “forward-looking feature” of string theory inside SMT-LIB 2.6, namely, the operator `to_re`, which converts a string (possibly with string variables) into a regular language. This results in a highly expressive language, which may capture word equations with Kleene stars. Existing solvers and benchmarks only handle the use cases of `to_re`, to which the input contains only string constants. Using parameters, we may capture constraints of the form  $x \in y^*$ , where  $x$  is a string variable and  $y$  is a “character variable” (meaning, string variable of length 1). This can be expressed as follows in SMT-LIB 2.6:

```
(declare-fun x () String)
(declare-fun w () String)
(assert (str.in_re w (re.* (str.to_re x) ) ) )
(assert (str.len x 1))
```

Third, we could also allow other effective boolean algebras, and consider instead *sequence theories*. Although such an extension is partly supported by leading SMT-solvers like Z3 [de Moura and Bjørner(2008)] and CVC5 [Barbosa et al.(2022)], there is as yet a standard logic and file format for sequence theories. In addition, the decidability of such theories has only been very recently studied [Jež et al.(2023)], whereby the quantifier-free fragment consisting of sequence equational constraints (i.e. concatenation of sequence variables and constants) and regular constraints (as parametric symbolic automata) is shown to be reducible to the case of finite alphabet, but incurring an exponential blow-up in the alphabet size. An example of such a constraint over LIA is

$$yz = zy \wedge y \in ([x \equiv_6 p])^+ \wedge z \in ([x \equiv_7 p^+]),$$

which has solutions  $y, z \mapsto \mathbb{Z}^*$ , which satisfy the equation  $yz = zy$  and that  $y$  (resp.  $z$ ) is a sequence of numbers that are  $p$  modulo 6 (resp.  $p$  modulo 7), for some  $p \in \mathbb{Z}$ . Our results allow us to also analyze such constraints by similar approach outlined above for string constraints, even when the sequence constraints are additionally extended with other predicates that we permit for string constraints (e.g. length constraints, contains, etc.) and symbolic context-free grammars.

## 5 Implementation

We implemented our approach described in Section 4 in C++. We used the Z3 [de Moura and Bjørner(2008)] library to parse and represent SMT-LIB formulas. We supported symbolic regular expressions by adapting the symbolic automata code and translations from the Z3 codebase. In the next sections we describe the optimisations we have implemented, the benchmarks used for testing, and then finally our results and analysis.

### 5.1 Optimisations

We improve the performance of the tool with two optimisations. The first reduces the number of characters  $a_i^p$  required for each transition label, the second helps restrict the search space of Z3 when solving the final abstracted constraints.

We remark that our implementation allows predicates

$$\psi(curr) := (curr = a) \quad \text{and} \quad \psi(curr) := str.in\_re(curr, re.in\_range(a, a')) .$$

In our optimisations, we do not exploit this interval representation. This means our optimisations apply to theories other than strings. In a dedicated string solver it would be possible to use well-known optimisations for character intervals to produce smaller formulas.

#### 5.1.1 Reducing the Characters per Transition Label

In the encoding above, each transition label requires  $2n \log(n)$  different character variables. However, consider the transition predicate  $p(curr) := (curr = a)$  that asserts that the character on the transition is the ‘a’ character. Clearly there is only one character that can satisfy  $p$  and  $2n \log(n)$  characters are not needed.

Similarly, if the only output predicate were  $\psi(curr) := \top$ , then all  $2n \log(n)$  characters contribute the same vector (1). In this case also only one character is required.

Using these observations, we approximate the number of characters than can satisfy  $p$  while having pairwise different profiles with respect to the output predicates they satisfy. To do this, we place the output predicates into “buckets”. Initially, one may suppose that each predicate is in its own bucket

$$\{\psi_1\}, \dots, \{\psi_n\} .$$

Supposing each character can either satisfy or not satisfy a predicate, a naive upper bound on the number of possible characters with different profiles is  $2 \times \dots \times 2 = 2^n$ .

However, suppose the first two output predicates  $\psi_1$  and  $\psi_2$  were such that there is no character  $a$  such that  $\psi_1(a) \wedge \psi_2(a)$  holds. That is, a character either satisfies  $\psi_1$  or  $\psi_2$  but never both. This gives three possibilities ( $a$  satisfies  $\psi_1$ ,  $a$  satisfies  $\psi_2$ , or  $a$  satisfies neither) instead of the naive upper bound  $2^2 = 4$ . In fact, this condition can be tightened: we only need that there is no character satisfying the transition predicate  $p$  that can simultaneously satisfy  $\psi_1$  and  $\psi_2$ .

We can extend this to multiple predicates. If  $\psi_1, \dots, \psi_{n'}$  are mutually exclusive (i.e. any value  $a$  can only satisfy at most one of the predicates), then there are  $n' + 1$  possibilities instead of  $2^{n'}$ .

When we identify such situations, we can replace the buckets  $\{\psi_1\}, \dots, \{\psi_{n'}\}$  with a single bucket  $\{\psi_1, \dots, \psi_{n'}\}$ .

Supposing we are able to group the output predicates into buckets  $B_1, \dots, B_{n'}$ . The number of possible vectors with respect to the predicates in a bucket  $B$  is the size of the bucket, plus

one if it's possible to simultaneously satisfy  $p$  and not satisfy any of the predicates in the bucket. That is, for each bucket, let

$$|B|_p := \begin{cases} |B| + 1 & \text{if } p(\text{curr}) \wedge \bigwedge_{\psi \in B} \neg \psi(\text{curr}) \text{ is satisfiable} \\ |B| & \text{otherwise.} \end{cases}$$

Our approximation of the upper bound on the number of characters for buckets  $B_1, \dots, B_{n'}$  is then

$$|B_1|_p \times \dots \times |B_{n'}|_p.$$

If this value is less than  $2n \log(n)$ , we use it instead of  $2n \log(n)$  for the characters associated with the transition label  $p$  in the encoding above.

To compute the buckets  $B_1, \dots, B_{n'}$  we use Algorithm 2 which is a simple greedy approach to allocating label predicates to buckets. Note, the “continue” keyword jumps to the next iteration of the for loop, so a new bucket is only created if a predicate overlaps with some predicate in all buckets computed so far.

---

**Algorithm 2:** ComputeBuckets( $p, \{\psi_1, \dots, \psi_n\}$ )

---

```

1 BList  $\leftarrow \varepsilon$ ;
2 for  $\psi$  in  $\{\psi_1, \dots, \psi_n\}$  do
3   for  $B$  in BList do
4     if  $p(\text{curr}) \wedge \psi(\text{curr}) \wedge \psi'(\text{curr})$  unsatisfiable for all  $\psi'$  in  $B$  then
5        $B \leftarrow B \cup \{\psi\}$ ;
6     continue
7   BList  $\leftarrow$  BList,  $\{\psi\}$ 
8 return BList

```

---

### 5.1.2 Restricting the Search Space

Our second optimisation is to help Z3 to determine the satisfiability of an abstracted formula. Suppose for some  $p$  we have character variables  $a_1^p, \dots, a_m^p$ . Suppose further that the solver has managed to determine that the assignment  $a_1, \dots, a_m$  cannot lead to a satisfying assignment. It is clear that any permutation of  $a_1, \dots, a_m$  also cannot lead to a satisfying assignment. However, without sophisticated inference, the solver needs to repeat the proof for all permutations. We extend our encoding to eliminate permutations as much as possible.

First, we assume the characters have a linear order  $<$ . That is, we can assert  $a < a'$ . In our implementation we represent characters with integers, so such an ordering is readily available. This means we can add the following constraint to our formula to eliminate permutations.

$$\bigwedge_{p \in \text{Labels}(\mathcal{A})} \bigwedge_{1 \leq i < 2n \log(n)} a_i^p < a_{i+1}^p$$

We can go a little further and also enforce that characters have different profiles with respect to the satisfaction of the output predicates. We enforce this with the following constraint.

$$\bigwedge_{p \in \text{Labels}(\mathcal{A})} \bigwedge_{1 \leq i < 2n \log(n)} \bigvee_{1 \leq j \leq n} \psi_j(a_i^p) \neq \psi_j(a_{i+1}^p)$$



Notice that we could have enforced this constraint for each pair of characters  $a_i, a_{i'}$  for  $1 \leq i \neq i' \leq 2n \log(n)$ . However, this would have required a much larger formula.

## 5.2 Experimental Results

Our implementation is written in C++ and uses Z3 4.12.1. Z3 is used to read SMT-LIB files and its data structures are used to represent the formulas. Z3 is also used as the backend solver for the produced constraints. Our implementation of symbolic automata is a slightly adapted version of the internal Z3 code.<sup>4</sup> We represent characters as Z3 integers, using their unsigned character codes. The implementation is available via its online repository [SymParikh Repository(2023)] and as an artifact with a disk image on Zenodo [SymParikh Artifact(2023)].

Our tool provides two methods for selecting the predicates to use in the Parikh image. In the default mode, the predicates are those appearing on the transitions of the symbolic automata constructed when parsing the input regular expressions. We only select those predicates that are of the form  $\varphi(curr) := curr = a$  or  $\varphi(curr) := str.in\_re(curr, re.in\_range(a, a'))$ . In the second mode, we additionally take predicates of the form  $\varphi(curr) := curr = a$  from the string literals appearing in the string equations. That is, if  $a_1 \dots a_n$  appears as a string literal in a string equation, we introduce the predicates  $\varphi_i(curr) := curr = a_i$  for all  $1 \leq i \leq n$ .

We describe the benchmarks used before giving the results.

### 5.2.1 Benchmark Sets

We used several benchmark sets from SMTCOMP 2022 [SMTCOMP2022(2022)], under the QF\_SLIA category. That is, quantifier-free constraints using strings and linear integer arithmetic. We also generated a set of benchmarks from regular expressions with a 5-star rating on [regexlib.com](https://regexlib.com).

Because we intend our technique to complement existing solvers, we restricted our attention to “difficult” benchmarks. We defined the “difficult” benchmarks to be those that could not be solved by Z3 in less than 10 seconds.

- The Norn benchmarks were introduced for the Norn tool [Abdulla et al.(2014)] and consist of concatenations of string literals and variables tested for membership (and non-membership) of regular expressions.
- The Kepler benchmarks were introduced for the Kepler tool [Pham et al.(2018)] and consist of quadratic word equations. That is, equality tests between two concatenations of string literals and variables, with each variable appearing at most twice.
- The WordEQ benchmarks were randomly generated by us for this paper from regular expressions taken from [regexlib.com](https://regexlib.com). This website collects user-submitted regular expressions for tasks such as email recognition, currency values, and others. We took regular expressions with a 5-star rating to avoid spam submissions.

The generated benchmarks were designed to test conjunctions of membership queries

---

<sup>4</sup>Specifically, we changed the representation to use transition sets instead of vectors to avoid transition duplication. We also used sets instead of vectors during minterm calculation when complementing automata to avoid multiple copies of the same predicate. Other minor changes include using Z3's push/pop feature instead of reset, and providing some extra convenience functions.

between overlapping regular expressions. We generated 100 benchmarks of the form

$$\begin{aligned} & \text{str.in\_re}(x_1, (r_1 \dots r_n)^*) \wedge \\ & \text{str.in\_re}(x_2, (r_1 \dots r_n)^+(r'_1 \dots r'_m)^+) \wedge \\ & \text{str.in\_re}(x_3, (r'_1 \dots r'_m)^*) \wedge \\ & s_1 = s_2 \end{aligned}$$

where  $1 \leq n, m \leq 3$  are randomly chosen integers,<sup>5</sup>  $r_1, \dots, r_n, r'_1, \dots, r'_m$  were randomly selected from the regular expressions obtained as above, and  $s_1$  and  $s_2$  are each concatenations of three variables picked randomly (possibly with duplicates) from  $\{x_1, x_2, x_3\}$  such that each variable appears at least once in  $s_1$  or  $s_2$  (and possibly in both).

In addition to the above sets, we also considered other QF<sub>SLIA</sub> benchmarks submitted to SMTCOMP 2022. These results are not included as the remaining benchmarks either contained unsupported features (such as string-to-integer functions, or character index functions), or were solved within 10s by Z3.

### 5.2.2 Comparison Solvers

We compare our tool with three state-of-the-art string solvers: Z3 (4.12.1) [de Moura and Bjørner(2008)], CVC5 (1.0.5) [Barbosa et al.(2022)], and OSTRICH [Chen et al.(2019), Chen et al.(2022), Chen et al.(2020), UVerifiers(2023)]. Z3 is a well-known SMT solver developed at Microsoft. CVC5 performs strongly in SMTCOMP competitions. Because the performance of Z3 and CVC5 can sometimes be similar, we also compare with two variants of the OSTRICH tool, which uses an automaton-based approach and often out-performed Z3 and CVC5 on unsat instances in SMTCOMP 2022. The CEA variant of OSTRICH uses cost-register automata and also makes use of Parikh images [Chen et al.(2020)]. It was run with the parameters `+parikh` and `-profile=strings`. Both variants were taken from the Cea-new branch of OSTRICH, commit ce855e26 [UVerifiers(2023)].

### 5.2.3 Results

Our experiments were performed on a Lenovo X380 Yoga ThinkPad with 8Gb RAM and 8 Intel® i7-8550U 1.8GHz CPUs, running Arch Linux (kernel 6.4.1). We used the default method for generating predicates for the Parikh image for all benchmarks except Kepler. For Kepler we extracted predicates from the string literals as the benchmarks do not contain regular expressions. When running the tools on the “difficult” benchmarks, we set the timeout to 30s.

Our tool over-approximates the true satisfiability of the input string equations and may return false-positives. Hence we are interested in the number of unsatisfiable instances, and those reported incorrectly as sat by our tool. For each benchmark set we consider the following.

- How many of the total number of benchmarks were “difficult”?
- How many of the “difficult” benchmarks were unsatisfiable instances?
- How many of the unsatisfiable instances were identified as false-positives by our tool (i.e. our tool returned “sat”)?
- The runtime of our solver, compared with other competitive solvers?

---

<sup>5</sup>Using Python’s `random.randint` function.

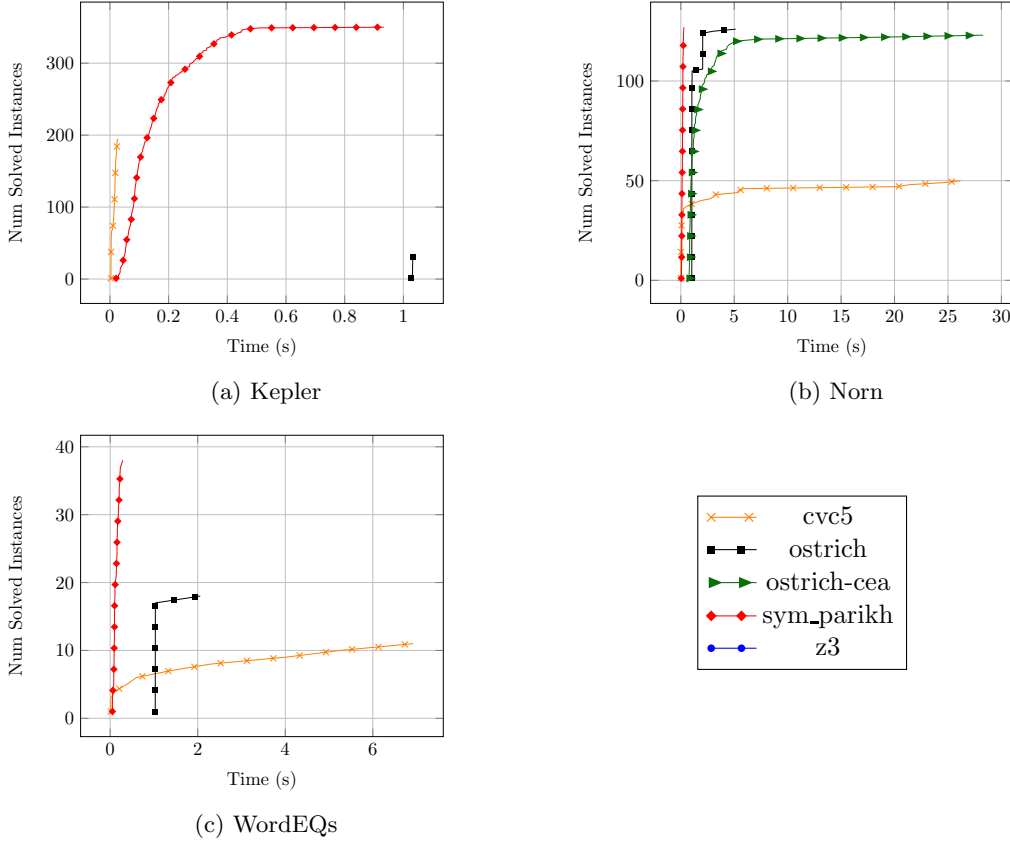


Figure 1: The number of instances solved in the given time across the benchmark sets. The line markings (shapes) are only to distinguish lines without colors, and are not individual data points. Our tool is labelled sym\_parikh.

Set	Instances	Difficult	Unsat	Approx Sat
Kepler	587	350	195	129
Norn	1027	127	127	9
WordEQs	100	38	35	0

Table 1: Summary data for benchmark sets. For each benchmark set, the table shows the number of instances, how many of the instances are “difficult”, the number of difficult benchmarks that are unsatisfiable, and the number of those unsatisfiable instances identified as satisfiable by our approximation.

The results for the final bullet point are presented in Figure 1, where our tool is labelled sym\_parikh. These graphs show the cumulative number of benchmarks solved in the given time. The remaining data is in Table 1. We note that Z3 did not solve any of the selected benchmarks within the timeout. However, we expect this is due to the bias in benchmark selection: we chose “difficult” benchmarks, where the Z3 solver was used to determine difficulty. Hence, only benchmarks that Z3 found difficult were included.

### 5.3 Analysis

Of 515 difficult instances, the majority – 357 (69%) – were found to be unsatisfiable. Of these, 219 (61%) were able to be proved unsatisfiable using the Parikh image abstraction. Thus, our approach gives useful results in 42% of considered cases. It can be seen in Figure 1 that performance was relatively robust on our benchmarks when compared with the exact analysis of the comparison tools, with all queries answered within 1s. This shows some potential for the use of the approach in the optimisation of string solvers. However, the performance can be seen to vary between the benchmarks sets. Since the summary results can be affected by the number of available benchmarks in each set, we discuss each set individually below. This will allow us to gain some intuition on where the approach may be best applied, and where it may be less useful.

Our tool performed well on the Norn benchmarks, solving all instances almost immediately. All difficult benchmarks were unsatisfiable instances. Our tool reported 9 false positives. These results are promising and indicate that the Parikh image abstraction may prove useful in quickly filtering unsatisfiable string constraints with regular expression containment checks.

The performance on the Kepler benchmarks was more mixed. These benchmarks proved difficult for most solvers, with only CVC5 and our tool able to return a large number of answers within the timeout period. CVC5 solved fewer instances than our tool, but did so more quickly. We note that our tool solved almost all instances within 0.5s. However, out of 195 unsatisfiable instances, our tool reported 129 false positives. We conjecture that this high false positive rate can be explained by the nature of the word equations. The values taken by the string variables were not limited by regular expression containment checks. This provides a lot of freedom for variables to take on values that equalise the Parikh images of both sides of the word equations, especially in cases where a variable only appears on one side of the equation. For example, in  $ax = xy$ , the variable  $y$  can contain the required  $a$  character. Unsatisfiability of Parikh image equality can require rarer inconsistencies. For example  $xy = yax$  will always require one more  $a$  character on the right hand side than the left.

Finally, our tool performed well on the difficult WordEQ instances. Of the 38 that were difficult for Z3, 35 were unsatisfiable instances. Our tool was able to determine the correct result quickly in all cases. This shows that the Parikh image abstractions may prove useful for examples containing complex interactions between overlapping regular expressions.

## 6 Conclusion

We have investigated Parikh images of languages over symbolic and parametric alphabets. In such a settings, the large, or even infinite alphabet makes a naive use of Parikh images impractical. Instead, our parametric version of the Parikh image is relative to a sequence of predicates  $\Psi = \psi_1, \dots, \psi_n$  and counts the number of times each predicate is satisfied by a character in the word.

The fact that Parikh images over classical context-free grammars can be computed by a linear-sized existential Presburger formula is a key ingredient in several verification applications. We introduce a parametric version of context-free grammars and an equivalent pushdown model.

Because the alphabet is large and multiple predicates can be satisfied simultaneously, one may expect an exponential blow-up over the classical results. Surprisingly, this turns out not to be the case. We can represent the Parikh image of a parametric context-free grammar with a polynomially-sized existential formula, and the complexity of related decision problems remains the same.

We presented an application of our results to overapproximate satisfiability of string constraints and provided an implementation based on Z3. Our experimental results showed that constraints that are difficult for existing solvers can be solved quickly using our abstraction.

*Future work.* These initial results suggest several avenues of future work. We first discuss limitations of our implementation. Firstly, our implementation makes a naive selection of predicates  $\Psi$  over which to compute the Parikh image. Improved predicate selection algorithms may balance the need for insightful information about the constraints being analysed, and the need to keep the number of variables small to allow constraints to be solved quickly. One may also investigate how existing solvers can deploy these techniques from within the solver, rather than as a one-off preprocessing step that analyses the whole formula at once. Secondly, our prototypical application to string constraint solving does not exploit the full potential of the results. For example, SMT-LIB does not currently support context-free constraints (as supported by some solvers like TRAU [Abdulla et al.(2018), Abdulla et al.(2017)]) and sequence theories over any effective boolean algebra [Jež et al.(2023)], and we have remarked that our results admit an easy extension to these. As an example, we may use parametric context-free languages to analyse streams of XML data, where the set of possible tags is infinite and should respect a nested structure.

On the theory side, it is still an open problem whether our results can be extended to other classes of recognizers over infinite alphabets, e.g., [D'Antoni et al.(2019), Brunet and Silva(2019), Moerman et al.(2017), Figueira and Lin(2022)]. In particular, we mention recent results on Parikh images of subclasses of nominal automata [Hofman et al.(2021)] and variants of data automata [Figueira and Lin(2022)], which provide a more precise Parikh abstraction and thus require a higher computational complexity (e.g. in [Figueira and Lin(2022)] double-exponential time algorithms). Secondly, in the light of the polynomial-time complexity result [Kopczynski and To(2010)] on reasoning about Parikh images of NFA with fixed alphabet size  $k$ , one could study Parikh images of parametric automata with a fixed number of predicates (for certain alphabets like ASCII, this number might be as small as 10 [Moseley et al.(2023), D'Antoni and Veanes(2021)]). Here, a simple application of the result in [Kopczynski and To(2010)] yields a polynomial-time complexity for any fixed  $k$ , but the actual complexity would be double exponential in  $k$ . Is it possible to lower this to a single exponential in  $k$ ?

Finally, one could investigate further potential applications of our results. For example, as explained in [D'Antoni and Veanes(2021)], model checking is typically done over Kripke structures over atomic propositions  $P_1, \dots, P_n$ . This gives rise as well to exponential-sized alphabets. Parikh's Theorem for symbolic automata could potentially be used to model checking temporal logics with additional predicate-counting abstractions. Similar applications for the case of finite alphabets have been discussed in [Hague and Lin(2011), Laroussinie et al.(2012), Laroussinie et al.(2010)]. To avoid potentially large automata, one could potentially also consider restrictions of temporal logics (e.g. LTL with only future/global operators [Benedikt et al.(2013)]).

**Acknowledgments** We thank anonymous reviewers, Nikolaj Bjorner, Oliver Markgraf, and Margus Veanes for helpful comments. Artur Jež was supported under National Science Centre, Poland project number 2017/26/E/ST6/00191. Anthony Lin was supported by European Research Council under European Union's Horizon research and innovation programme (grant agreement no 101089343).

## References

- [Abdulla et al.(2017)] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. 2017. Flatten and conquer: a framework for efficient analysis of string constraints. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 602–617. <https://doi.org/10.1145/3062341.3062384>
- [Abdulla et al.(2018)] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. 2018. Trau: SMT solver for string constraints. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj S. Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–5. <https://doi.org/10.23919/FMCAD.2018.8602997>
- [Abdulla et al.(2014)] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String Constraints for Verification. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 150–166. [https://doi.org/10.1007/978-3-319-08867-9\\_10](https://doi.org/10.1007/978-3-319-08867-9_10)
- [Abdulla et al.(2019)] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukás Holík, and Petr Janku. 2019. Chain-Free String Constraints. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11781)*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer, 277–293. [https://doi.org/10.1007/978-3-030-31784-3\\_16](https://doi.org/10.1007/978-3-030-31784-3_16)
- [Amadini et al.(2019)] Roberto Amadini, Mak Andrlon, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2019. Constraint Programming for Dynamic Symbolic Execution of JavaScript. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11494)*, Louis-Martin Rousseau and Kostas Stergiou (Eds.). Springer, 1–19. [https://doi.org/10.1007/978-3-030-19212-9\\_1](https://doi.org/10.1007/978-3-030-19212-9_1)
- [Barbosa et al.(2022)] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. [https://doi.org/10.1007/978-3-030-99524-9\\_24](https://doi.org/10.1007/978-3-030-99524-9_24)
- [Barceló et al.(2012)] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Trans. Database Syst.* 37, 4 (2012), 31:1–31:46. <https://doi.org/10.1145/2389241.2389250>
- [Barner(2006)] S. Barner. 2006. *H3 mit Gleichheitstheorien*. Diploma Thesis. Technische Universität München.
- [Barrett et al.(2017)] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa. Available at [www.SMT-LIB.org](http://www.SMT-LIB.org).
- [Benedikt et al.(2013)] Michael Benedikt, Rastislav Lenhardt, and James Worrell. 2013. LTL Model Checking of Interval Markov Chains. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013*.



- Proceedings (Lecture Notes in Computer Science, Vol. 7795)*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 32–46. [https://doi.org/10.1007/978-3-642-36742-7\\_3](https://doi.org/10.1007/978-3-642-36742-7_3)
- [Blahoudek et al.(2023)] Frantisek Blahoudek, Yu-Fang Chen, David Chocholatý, Vojtech Havlena, Lukás Holík, Ondrej Lengál, and Juraj Síc. 2023. Word Equations in Synergy with Regular Constraints. In *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14000)*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.). Springer, 403–423. [https://doi.org/10.1007/978-3-031-27481-7\\_23](https://doi.org/10.1007/978-3-031-27481-7_23)
- [Bouajjani et al.(2003)] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. 2003. A generic approach to the static analysis of concurrent programs with procedures. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 62–73. <https://doi.org/10.1145/604131.604137>
- [Brunet and Silva(2019)] Paul Brunet and Alexandra Silva. 2019. A Kleene Theorem for Nominal Automata. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece (LIPIcs, Vol. 132)*, Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 107:1–107:13. <https://doi.org/10.4230/LIPIcs.ICALP.2019.107>
- [Chen et al.(2022)] Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2022. Solving string constraints with Regex-dependent functions through transducers with priorities and variables. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. <https://doi.org/10.1145/3498707>
- [Chen et al.(2020)] Taolue Chen, Matthew Hague, Jinlong He, Denghang Hu, Anthony Widjaja Lin, Philipp Rümmer, and Zhilin Wu. 2020. A Decision Procedure for Path Feasibility of String Manipulating Programs with Integer Data Type. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*. 325–342. [https://doi.org/10.1007/978-3-030-59152-6\\_18](https://doi.org/10.1007/978-3-030-59152-6_18)
- [Chen et al.(2019)] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2019. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.* 3, POPL (2019), 49:1–49:30. <https://doi.org/10.1145/3290362>
- [Christensen et al.(2003)] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*, Radhia Cousot (Ed.). Springer, 1–18. [https://doi.org/10.1007/3-540-44898-5\\_1](https://doi.org/10.1007/3-540-44898-5_1)
- [Daca et al.(2016)] Przemyslaw Daca, Thomas A. Henzinger, and Andrey Kupriyanov. 2016. Array Folds Logic. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 230–248. [https://doi.org/10.1007/978-3-319-41540-6\\_13](https://doi.org/10.1007/978-3-319-41540-6_13)
- [D’Antoni and Alur(2014)] Loris D’Antoni and Rajeev Alur. 2014. Symbolic Visibly Pushdown Automata. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 209–225. [https://doi.org/10.1007/978-3-319-08867-9\\_14](https://doi.org/10.1007/978-3-319-08867-9_14)
- [D’Antoni et al.(2019)] Loris D’Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva. 2019. Symbolic Register Automata. In *CAV, Isil Dillig and Serdar Tasiran (Eds.)*, Vol. 11561. Springer, 3–21. [https://doi.org/10.1007/978-3-030-25540-4\\_1](https://doi.org/10.1007/978-3-030-25540-4_1)
- [D’Antoni and Veanes(2017)] Loris D’Antoni and Margus Veanes. 2017. The Power of Symbolic Automata and Transducers. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in*

- Computer Science, Vol. 10426*), Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 47–67. [https://doi.org/10.1007/978-3-319-63387-9\\_3](https://doi.org/10.1007/978-3-319-63387-9_3)
- [D’Antoni and Veanes(2021)] Loris D’Antoni and Margus Veanes. 2021. Automata modulo theories. *Commun. ACM* 64, 5 (2021), 86–95. <https://doi.org/10.1145/3419404>
- [David et al.(2012)] Claire David, Leonid Libkin, and Tony Tan. 2012. Efficient reasoning about data trees via integer linear programming. *ACM Trans. Database Syst.* 37, 3 (2012), 19:1–19:28. <https://doi.org/10.1145/2338626.2338632>
- [de Moura and Bjørner(2008)] L. Mendonça de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340.
- [Eisenbrand and Shmonin(2006)] Friedrich Eisenbrand and Gennady Shmonin. 2006. Carathéodory bounds for integer cones. *Operations Research Letters* 34, 5 (2006), 564–568. <https://doi.org/10.1016/j.orl.2005.09.008>
- [Esparza(1997)] Javier Esparza. 1997. Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes. *Fundam. Informaticae* 31, 1 (1997), 13–25. <https://doi.org/10.3233/FI-1997-3112>
- [Esparza and Ganty(2011)] Javier Esparza and Pierre Ganty. 2011. Complexity of pattern-based verification for multithreaded programs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 499–510. <https://doi.org/10.1145/1926385.1926443>
- [Faran and Kupferman(2020)] Rachel Faran and Orna Kupferman. 2020. On Synthesis of Specifications with Arithmetic. In *SOFSEM 2020: Theory and Practice of Computer Science - 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20-24, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12011)*, Alexander Chatzigeorgiou, Riccardo Dondi, Herodotos Herodotou, Christos A. Kapoutsis, Yannis Manolopoulos, George A. Papadopoulos, and Florian Sikora (Eds.). Springer, 161–173. [https://doi.org/10.1007/978-3-030-38919-2\\_14](https://doi.org/10.1007/978-3-030-38919-2_14)
- [Figueira et al.(2022)] Diego Figueira, Artur Jež, and Anthony W. Lin. 2022. Data Path Queries over Embedded Graph Databases. In *PODS ’22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. 189–201. <https://doi.org/10.1145/3517804.3524159>
- [Figueira and Lin(2022)] Diego Figueira and Anthony Widjaja Lin. 2022. Reasoning on Data Words over Numeric Domains. In *LICS ’22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*. 37:1–37:13. <https://doi.org/10.1145/3531130.3533354>
- [Gange et al.(2015)] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2015. A Tool for Intersecting Context-Free Grammars and Its Applications. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9058)*, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 422–428. [https://doi.org/10.1007/978-3-319-17524-9\\_31](https://doi.org/10.1007/978-3-319-17524-9_31)
- [Gange et al.(2016)] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2016. A complete refinement procedure for regular separability of context-free languages. *Theor. Comput. Sci.* 625 (2016), 1–24. <https://doi.org/10.1016/j.tcs.2016.01.026>
- [Grumberg et al.(2010)] Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. 2010. Variable Automata over Infinite Alphabets. In *Language and Automata Theory and Applications, 4th International Conference, LATA 2010, Trier, Germany, May 24-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6031)*, Adrian-Horia Dediu, Henning Fernau, and Carlos Martín-Vide (Eds.). Springer, 561–572. [https://doi.org/10.1007/978-3-642-13089-2\\_47](https://doi.org/10.1007/978-3-642-13089-2_47)
- [Hague and Lin(2011)] Matthew Hague and Anthony Widjaja Lin. 2011. Model Checking Recur-

- sive Programs with Numeric Data Types. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 743–759. [https://doi.org/10.1007/978-3-642-22110-1\\_60](https://doi.org/10.1007/978-3-642-22110-1_60)
- [Hague and Lin(2012)] Matthew Hague and Anthony Widjaja Lin. 2012. Synchronisation- and Reversal-Bounded Analysis of Multithreaded Programs with Counters. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 260–276. [https://doi.org/10.1007/978-3-642-31424-7\\_22](https://doi.org/10.1007/978-3-642-31424-7_22)
- [Hofman et al.(2021)] Piotr Hofman, Marta Jucepczuk, Slawomir Lasota, and Mohnish Pattathurajan. 2021. Parikh's theorem for infinite alphabets. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–13. <https://doi.org/10.1109/LICS52264.2021.9470626>
- [Janku and Turonová(2019)] Petr Janku and Lenka Turonová. 2019. Solving String Constraints with Approximate Parikh Image. In *Computer Aided Systems Theory - EUROCAST 2019 - 17th International Conference, Las Palmas de Gran Canaria, Spain, February 17-22, 2019, Revised Selected Papers, Part I (Lecture Notes in Computer Science, Vol. 12013)*, Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia (Eds.). Springer, 491–498. [https://doi.org/10.1007/978-3-030-45093-9\\_59](https://doi.org/10.1007/978-3-030-45093-9_59)
- [Jež et al.(2023)] Artur Jež, Anthony W. Lin, Oliver Markgraf, and Philipp Rümmer. 2023. Decision Procedures for Sequence Theories. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13965)*, Constantin Enea and Akash Lal (Eds.). Springer, 18–40. [https://doi.org/10.1007/978-3-031-37703-7\\_2](https://doi.org/10.1007/978-3-031-37703-7_2)
- [Kopczynski and To(2010)] Eryk Kopczynski and Anthony Widjaja To. 2010. Parikh Images of Grammars: Complexity and Applications. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*. 80–89. <https://doi.org/10.1109/LICS.2010.21>
- [Kozen(1997)] Dexter C. Kozen. 1997. *Automata and Computability*. Springer.
- [Kroening and Strichman(2008)] Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures*. Springer.
- [Laroussinie et al.(2010)] François Laroussinie, Antoine Meyer, and Eudes Pettonnet. 2010. Counting LTL. In *TIME 2010 - 17th International Symposium on Temporal Representation and Reasoning, Paris, France, 6-8 September 2010*, Nicolas Markey and Jef Wijsen (Eds.). IEEE Computer Society, 51–58. <https://doi.org/10.1109/TIME.2010.20>
- [Laroussinie et al.(2012)] François Laroussinie, Antoine Meyer, and Eudes Pettonnet. 2012. Counting CTL. *Log. Methods Comput. Sci.* 9, 1 (2012), 1–34. [https://doi.org/10.2168/LMCS-9\(1:3\)2013](https://doi.org/10.2168/LMCS-9(1:3)2013)
- [Lin and Barceló(2016)] Anthony Widjaja Lin and Pablo Barceló. 2016. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 123–136. <https://doi.org/10.1145/2837614.2837641>
- [Long et al.(2012)] Zhenyue Long, Georgel Calin, Rupak Majumdar, and Roland Meyer. 2012. Language-Theoretic Abstraction Refinement. In *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7212)*, Juan de Lara and Andrea Zisman (Eds.). Springer, 362–376. [https://doi.org/10.1007/978-3-642-28872-2\\_25](https://doi.org/10.1007/978-3-642-28872-2_25)
- [Loring et al.(2019)] Blake Loring, Duncan Mitchell, and Johannes Kinder. 2019. Sound regular expression semantics for dynamic symbolic execution of JavaScript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 425–438.

- <https://doi.org/10.1145/3314221.3314645>
- [Minamide(2005)] Yasuhiko Minamide. 2005. Static approximation of dynamically generated Web pages. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, Allan Ellis and Tatsuya Hagino (Eds.). ACM, 432–441. <https://doi.org/10.1145/1060745.1060809>
- [Moerman et al.(2017)] Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michal Szynwelski. 2017. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 613–625. <https://doi.org/10.1145/3009837.3009879>
- [Moseley et al.(2023)] Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. 2023. Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1026–1049. <https://doi.org/10.1145/3591262>
- [Nelson and Oppen(1979)] Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 2 (oct 1979), 245–257. <https://doi.org/10.1145/357073.357079>
- [Parikh(1966)] Rohit Parikh. 1966. On Context-Free Languages. *J. ACM* 13, 4 (1966), 570–581. <https://doi.org/10.1145/321356.321364>
- [Pham et al.(2018)] Long H. Pham, Quang Loc Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin. 2018. Testing heap-based programs with Java StarFinder. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 268–269. <https://doi.org/10.1145/3183440.3194964>
- [Raya(2023)] Rodrigo Raya. 2023. *The Complexity of Checking Non-Emptiness in Symbolic Tree Automata*. Retrieved October 25, 2023 from <https://infoscience.epfl.ch/record/304426>
- [Saxena et al.(2010)] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*. IEEE Computer Society, 513–528. <https://doi.org/10.1109/SP.2010.38>
- [Sipser(2013)] Michael Sipser. 2013. *Introduction to the Theory of Computation* (third ed.). Course Technology, Boston, MA.
- [SMTCOMP2022(2022)] SMTCOMP2022 2022. *The International Satisfiability Modulo Theories (SMT) Competition 2022*. <https://smt-comp.github.io/2022/> Accessed: 7 July 2023.
- [Stanford et al.(2021)] Caleb Stanford, Margus Veanes, and Nikolaj S. Bjørner. 2021. Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 620–635. <https://doi.org/10.1145/3453483.3454066>
- [SymParikh Artifact(2023)] SymParikh Artifact 2023. *Parikh's Theorem Made Symbolic: Artifact*. <https://zenodo.org/records/8417439> Accessed: 23 October 2023.
- [SymParikh Repository(2023)] SymParikh Repository 2023. *Parikh's Theorem Made Symbolic*. <https://gitlab.cim.rhul.ac.uk/uxac009/symparikh> Accessed: 23 October 2023.
- [To(2009)] Anthony Widjaja To. 2009. Model Checking FO(R) over One-Counter Processes and beyond. In *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5771)*, Erich Grädel and Reinhard Kahle (Eds.). Springer, 485–499. [https://doi.org/10.1007/978-3-642-04027-6\\_35](https://doi.org/10.1007/978-3-642-04027-6_35)
- [UUVifiers(2023)] UUVifiers 2023. *OSTRICH*. <https://github.com/uuverifiers/ostrich> Accessed: 12 June 2023.

- [Veanes et al.(2012)] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjorner. 2012. Symbolic Finite State Transducers: Algorithms and Applications. *SIGPLAN Not.* 47, 1 (jan 2012), 137–150. <https://doi.org/10.1145/2103621.2103674>
- [Verma et al.(2005)] Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. 2005. On the Complexity of Equational Horn Clauses. In *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3632)*, Robert Nieuwenhuis (Ed.). Springer, 337–352. [https://doi.org/10.1007/11532231\\_25](https://doi.org/10.1007/11532231_25)